Subverting Cryptographic Hardware used in Blockchain Consensus

Pratyush Ranjan Tiwari, Matthew Green

Johns Hopkins University

{pratyush, mgreen}@cs.jhu.edu

Abstract. In this work, we study and formalize security notions for algorithm substitution attacks (ASAs) on *cryptographic puzzles*.[1] Puzzles are difficult problems that require an investment of computation, memory, or some other related resource. They are heavily used as a building block for the consensus networks used by cryptocurrencies. These include primitives such as proof-of-work, proof-of-space, and verifiable delay functions (VDFs). Due to economies of scale, these networks increasingly rely on a small number of companies to construct opaque hardware or software (*e.g.,* GPU or FPGA images): this dependency raises concerns about cryptographic subversion. We first explore the threat model for these systems and then propose concrete attacks that (1) selectively reduce a victim's solving capability (*e.g.,* hashrate) and (2) exfiltrate puzzle solutions to an attacker. Our findings reveal that these devices could be subverted today, and detecting some attack variants is extremely hard and costly. We then suggest defenses, several of which can be applied to existing cryptocurrency hardware with minimal changes. We also discover that mining devices for many major proof-of-work cryptocurrencies already demonstrate errors exactly how a potentially subverted device would. Given that these attacks are relevant to all proof-of-work cryptocurrencies that have a combined market capitalization of around a few hundred billion dollars (2023), we recommend that all vulnerable mining protocols consider making the suggested adaptations today.

1 Introduction

Security for cryptographic systems depends on the existence of a trusted implementation. Unfortunately, recent experience shows that implementers cannot always be trusted. Examples of cryptographic subversion are increasingly common, ranging from backdoors in cryptocurrency wallets [59,52] to software packages [2] and smart contracts [53]. In several famous examples, highly motivated attackers surreptitiously modified the implementation of cryptographic algorithms as a means to subvert the operation of security systems that depend on them [16,22,1]. In the research literature such attacks are known as *algorithm-substitution attacks* (ASAs) and have been the subject of much formal

[1] A full version of this work is available at `https://eprint.iacr.org/2022/477`

study [61,11,10,14,31,23,33,15,47,48,46]. In an algorithm substitution attack, an attacker modifies a cryptographic implementation to replace a cryptographic algorithm or protocol with an adversarial replacement. The subverted device is then adopted by an honest user who is unaware of the substitution.

Cryptographic Puzzles. *Puzzles* are difficult problems that require an investment of computation, memory, or some other related resource [32]. Puzzles were originally proposed for spam prevention or avoidance of DoS attacks [34] but have more recently been adopted by cryptocurrencies such as Bitcoin, Ethereum, Chia and Filecoin for the purpose of *mining*, i.e., achieving consensus and minting new coins [4,28]. In each of these systems, solvers reap substantial economic rewards from solving puzzles. The dark side of this arrangement is that these financial incentives provide strong motivation for the development of sophisticated attacks [38].[2] We consider the problem of implementing and mitigating ASAs against cryptographic puzzle-solving hardware and software. The field of cryptocurrency mining is uniquely vulnerable to these attacks due to the fact that nearly all popular mining hardware (and much software) is manufactured by a small number of companies and that many of these companies also operate their own competing mining operations (see Figure 3) [58,7]. Cryptographic puzzle algorithms are also significantly different from the algorithms considered in much previous work on ASAs and therefore support both different attacks and defenses. Most notably, cryptographic puzzles typically do not make use of secret keys.[3]

Attack setting: cryptocurrency mining. While ASAs can be applied to any application of puzzles, in this work, a key topic of focus is the setting of cryptocurrency mining. These operations typically use large collections of puzzle-solving devices to evaluate solutions and thus mine new blocks in a cryptocurrency network. For many proof-of-work networks like Bitcoin, the most popular mining device (and possibly the only profitable ones) are specialized mining ASICs (Application-Specific Integrated Circuits) that are typically purchased inside an enclosure. Other systems rely on FPGA or GPU devices that use specialized implementations [54]: while these implementations can technically be reviewed by experts, the cost of these specialized reviews is often prohibitive. In addition to the purchase price of the hardware and software, the potential victim must also contribute substantial resources in the form of electricity (for proof-of-work) or other resources such as RAM or hard disk drives (for memory-hard proof-of-work [55] and proof-of-space constructions [29,8,25].) Mining operations can be organized in various configurations, two of which we

[2] While some examples of cryptographic puzzles such as proof-of-work have historically had a negative environmental impact through extremely high energy consumption, not all [29] cryptographic puzzles share this downside.

[3] A significant exception to this pattern is the work of Russell et al. [48], which focuses on correcting subverted hash functions (random oracles), with applications to proof-of-work. We focus on the more general primitive of puzzles, which can be built from hash functions, but may also be built from alternative ingredients (e.g., algebraic VDFs.)

consider in this work: (1) *Individual mining.* In the most traditional setting, a cryptocurrency node operated by the miner obtains puzzle instances by running the cryptocurrency network's software on a computer. The challenge for a subversion attacker in this setting is that a remote attacker may not have direct access to send and receive messages to/from the subverted devices: all communication with the device must be in the form of correctly-structured puzzle inputs and solutions that pass through the cryptocurrency network. (2) *Pool mining.* A second popular configuration allows mining devices to participate in a *mining pool* with many other miners [39]. Mining pools allow miners to distribute the risk of mining operations, improving the predictability of mining rewards. The victim miner (or the subverted device itself) receives puzzle instances from a central *pool coordinator*: pessimistically, this coordinator may collude with the subversion attacker.

Related work. Previous work on ASAs [13,11,23,15,24] focuses on algorithms that employ secret keys, such as decryption and digital signing algorithms. In this setting, the attacker's primary goal is usually to exfiltrate cryptographic secrets from the device: indeed, some proposed defenses have been aimed solely at increasing the cost of this exfiltration [12]. A series of insightful works by Russel, Tang, Yung and Zhou examined cryptographic algorithms that do not employ secret keys [47,46,48]: ASAs and defenses for primitives such as hash functions have been studied in great detail by these works. While these works provide defenses for subverted hash functions and random oracles (and mention cryptocurrency mining as an application), our work focuses on puzzle schemes as the base-level primitive. The high-level subversion philosophy for all our attacks is inspired by these works and previous work on ASAs. However, addressing the problem from a systems-oriented perspective, rather than focusing on the subversion of specific cryptographic hash functions, has enabled us to provide concrete analysis on whether *such attackers can exist for blockchains today and that they can indeed reap great monetary benefits from mounting these attacks.* Our is not the first work to consider the problem of re-designing puzzles to resist certain types of collusion. The work of Miller *et al.* [40] proposes *scratch-off puzzles*: these make mining pools obsolete by ensuring that the puzzles have a property making them "non-outsourceable" so that the mining operator wouldn't know if one of the members of the pool found a solution.

1.1 Our contributions

The most important concern that our work aims to address, which surprisingly, has not been shown previously is the following: we find that in many existing networks, subversion attacks can run for an extended period before any defender can be confident that an attack is underway. Moreover, real-world devices have various features that can make such attacks even more difficult to detect. For example, popular ASIC-based puzzle-solving devices routinely produce unexplained hardware failures on a small fraction of inputs. While these errors (Fig 1) can occur due to overheating and manufacturing errors, ***they are also pre-***

cisely the symptom one might expect from a subverted device.[4] These findings motivate better techniques to detect and prevent subversion attacks in deployed systems. More concretely, our contributions are as follows:

Antminer S19 Pro

Number of hash boards	Number of chips	Number of hardware errors	Operating frequency
1	114	108	525
2	114	982	525
3	114	132	525

Fig. 1: Hardware errors as they appear to a miner. These errors in mining devices reflect the number of inputs on which the mining ASIC fails to compute. Via real-world mining software [41], with very little information on why the error occurred and on what particular input.

1. We provide a simplified and general definition of cryptographic puzzles that suffices to analyze ASA attacks on real-world examples of cryptographic puzzles such as proof-of-work, verifiable delay functions, and proof of space protocols.
2. We formalize security notions for algorithm-substitution attacks (ASAs) on cryptographic puzzle schemes. We next analyze which attacks succeed in the real world. Furthermore, we extend previous ASA definitions to our setting by considering oracles that allow timing and resource usage of operations to be reported.
3. We devise realistic threat models under which we propose two different algorithm-substitution attacks on cryptographic puzzles. These include *load-shedding attacks*, which affects the puzzle-solving abilities of subverted devices on an unpredictable set of subverted inputs set by the attacker and *leeching attacks* in which a subverted device exfiltrates puzzle solutions to an attacker. Notably, our attack strategy can be combined to lower the miner hashrate required for the selfish mining strategy [30] to succeed (see Figure 4).
4. Using our framework, we provide concrete estimates for attack success probabilities and costs for the Bitcoin network. Since this is the most widely-deployed application of cryptographic puzzles in the real world. We demonstrate how such an attacker can benefit from these two algorithm-substitution attacks when deployed against Bitcoin's proof-of-work mining.

[4] See and Appendix G.1 for examples.

5. Finally, we provide countermeasures that, if appropriately utilized, render many of our proposed attacks ineffective. We also settle the question of whether device testing can help detect subversion in this setting. Our countermeasures to protect existing implementations can be instantiated with no changes to the underlying puzzle or consensus networks and at a minimal cost.
6. We implement one of our proposed countermeasures, an efficient masking protocol for Pietrzak's repeating-squaring VDFs. The resulting overhead is a miniscule fraction of the VDF evaluation time.

2 Technical Overview

We now provide an overview of our model and attacks. We then focus all of Section 3 on providing countermeasures against these attacks.
Formalizing puzzles. Our first objective is to re-formalize the notion of a *cryptographic puzzle*. While previous works have offered such formalization, they have in the past been tightly coupled to specific constructions such as the proof-of-work hash puzzles used by Bitcoin. Our goal is to identify a formalization that can be applied to a much broader category of puzzles used in practical systems. Our definitions simplify the definitions from the work of Groza and Warinschi [32]. However, in our formalization, we generalize each of these systems to a puzzle defined over a specific *resource*: considering both the minimum amount and the average amount of the puzzle resource required to find a valid puzzle solution is determined by a difficulty parameter. This allows us to capture the abstraction of many cryptographic puzzles of interest. In Appendix A, we formalize the security properties, and we demonstrate the utility of this definition by showing that many important primitives such as proof-of-work, VDFs, and proof-of-space fit the definitions of a cryptographic puzzle as described in Appendix A. Our puzzle definition is captured by the following PPT algorithms:

- $\mathsf{Setup}(1^\lambda, \Delta) \rightarrow \mathsf{pp}$: The puzzle generation algorithm takes as input a security parameter 1^λ and a difficulty parameter Δ. It outputs public parameters pp which fix the domain of the unprocessed input $\mathcal{X}_{\mathsf{Pre}}$, pre-processed input domain $\mathcal{X}$, range $\mathcal{Y}$ of the puzzle and other information required to compute a puzzle or verify a puzzle solution. All the following algorithms implicitly take pp as an input.
- $\mathsf{Pre}(x', \mathsf{aux}) \rightarrow x$: The puzzle pre-processing algorithm (optional) takes a puzzle input x' from the unprocessed input domain $\mathcal{X}_{\mathsf{Pre}}$ and an auxiliary input aux. It outputs a processed puzzle input $x \in \mathcal{X}$. If no pre-processing options are presented, this is just an identity map where $x' = x$ and $\mathcal{X}_{\mathsf{Pre}} = \mathcal{X}$.
- $\mathsf{Eval}(x, \mathsf{aux}) \rightarrow y/\perp$: The puzzle evaluation algorithm takes an input x from the pre-processed input domain and the auxiliary information aux which was used for pre-processing. It outputs a puzzle solution y if there exists a valid solution for the input, auxiliary input pair (x, aux), otherwise outputs $\perp$.

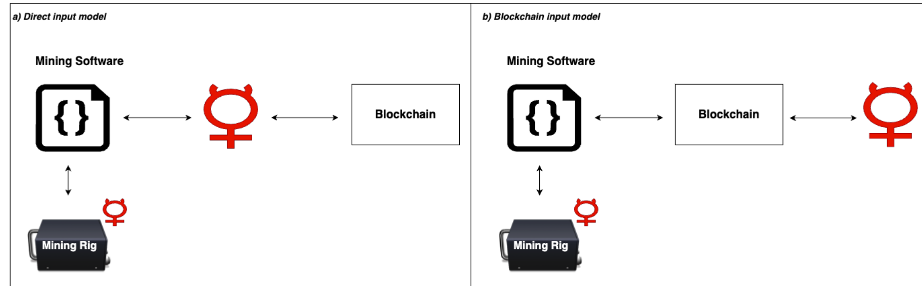


Fig. 2: a) **Direct input model.** b) **Blockchain input model.**

– $\mathsf{Verify}(x, \mathsf{aux}, y) \rightarrow 0/1$: The puzzle verification algorithm takes a pre-processed puzzle input x, the auxiliary information aux which was used for pre-processing and an input from the range y. It outputs either 0 or 1.

Defining attacker goals. The real-world examples of these puzzles are deployed in settings where puzzle evaluators reap the rewards from solving these puzzles. In many cases (such as sequential work puzzles), the fastest evaluator to find the puzzle solution is the only one who receives the reward. Therefore, we focus on attacks with the following goals: either to force the subverted evaluation hardware/software to malfunction/fail on certain puzzle instances, or else to exfiltrate solutions to the attacker. In all attack settings, we require that such subversion should not be detected by the owner of the hardware during testing: we realize this by postulating a remote attacker that may *initiate* an attack either by providing malicious puzzle instances, or else by providing some alternative "trigger" to the hardware. Once an attack has begun, we further consider the attacker's ability to detect that attack. We note that for many deployed puzzles, attack detection is challenging and may involve some statistical uncertainty, and that moreover, an attacker may reap substantial rewards from the attack before a defender can be certain that one is underway.

Threat model. We now define our threat model for real-world attacks. We consider two settings. In each one, a remote attacker interfaces with a subverted device (or devices) in order to trigger the attack and (optionally) receive exfiltrated results. The key difference between our two settings is in how a remote attacker communicates with the device: (i) *Direct input model.* In the first setting, a remote attacker has the ability to feed puzzle inputs to the victim's subverted mining device directly. This model captures hardware that is connected to a *mining pool in which the operator is under the attacker's control.* Why this setting is a completely reasonable one is easily demonstrated by our analysis in Figure 3 (ii) *Blockchain input model.* In this model the remote attacker must communicate with the subverted hardware via puzzle instances read/written from the chain (see Figure 2 for an illustration.)

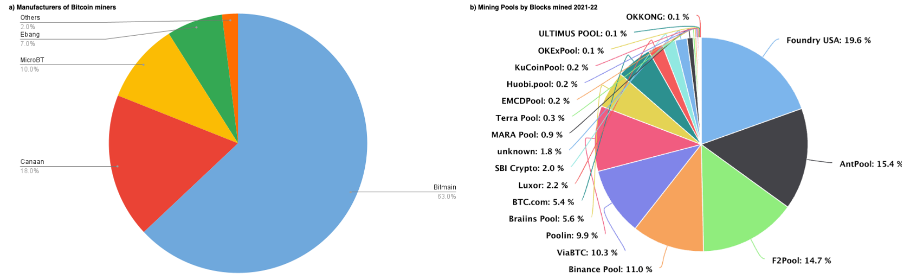


Fig. 3: Left: Market Share for Bitcoin mining ASIC manufacturers [44] and Right: Bitcoin blocks mined by various mining pools in 2021-22 [21]. AntPool (15.4%) and BTC.com (5.4%) together account for the highest control of hashpower and are run by the largest manufacturer (Bitmain).

2.1 Security against ASAs

When an ASA is mounted, an attacker replaces the standard algorithms $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ with the subverted algorithms $\overline{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$. The attacker goals are as described earlier.Subversion in this setting refers to the adversarial goal of $\mathcal{A}$ to reduce the puzzle-solving ability of $\mathcal{D}$, and the goal of $\mathcal{D}$ is to detect if it's running a subverted version of the algorithm Eval to solve the puzzle. Following the footsteps of other works on ASAs we define the goals of the subverting adversary/attacker $\mathcal{A}$ via the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\overline{\mathsf{P}}}$ and the detecting adversary/detector $\mathcal{D}$ via the detection game $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$. We first describe the detection game $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$ §B played between a challenger and a detecting adversary $\mathcal{D}$. Our oracles also output a counter cnt indicating the time taken to compute the response, unlike previous work on ASAs, this adds another challenge for the attacker. The subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\overline{\mathsf{P}}}$ §B is played between a challenger and a subverting adversary $\mathcal{A}$. This game captures the notion that $\mathcal{A}$ can easily find out if a device is subverted, simply because it crafted the subversion.

On the other hand, a detector is motivated to check its puzzle-solving device for possible subversion. The offline detection game §B and Definition 5 captures this notion for when the detector checks its device for subversion before it is used online. To capture the notion that a detector might stay vigilant and check device behavior when the device is puzzle-solving "online", we describe an online detection game $\mathbf{Det}+^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$ §B and Definition 6 which aims at capturing the attacker-subverted device interaction when the device is online and the attacker is trying to leverage the subversion. With these definitions, we then define security against ASAs.[5]

---

[5] **The role of state.** There might be attacks that leverage the use of state in the subverted puzzle algorithms. Since our model assumes a malicious manufacturer for the puzzle-solving hardware/software, the only algorithm where state can be utilized for attacks is the subverted evaluation algorithm $\overline{\mathsf{Eval}}$. Therefore, we term an attack

Definition 1 (Security against ASAs). *A cryptographic puzzle scheme* $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ *is considered secure against algorithm-substitution attacks if for all possible subversions* $\overline{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$ *of* P *if the scheme is subversion resistant against the attack in the sense of definition 4 or the attack is detectable in the sense of definition 5 (the scheme is only* ***online secure*** *if the attack is only online detectable in the sense of definition 6).*[6]

2.2 Attack strategies

Based on our threat models, we propose two attack strategies as described below. These attack strategies are valid even if the devices are actively subverted from the time of purchase. However, we also focus on variants of the attacks that are *input-induced*: this means that the malicious manufacturer/attacker uses trigger puzzle instances to send the puzzle solving/mining device into "subverted" mode. The benefit of this approach is that *prior to receiving any of these inputs* the device will behave indistinguishably from an honest implementation, and can therefore hide its nature until the attacker desires. Following the receipt of a valid trigger, the device enters an attack mode that may persist indefinitely or for a finite duration set by the attacker.[7] We identify two specific forms of attack that can be initiated by this trigger:

Load-Shedding Attack.(§C.1, §C.3) When the device encounters an attacker-formulated trigger, it silently rejects valid puzzle solutions according to a pre-specified strategy[8]. If the subverted devices make up a significant fraction of the overall network hashrate, this can reduce the overall hashrate and consequently increase the fraction of total hashrate that is provided by unsubverted devices (see Figure 4), this strategy lowers the selfish mining [30] threshold for the attacker as well. Theorems 5, 6 prove that the load-shedding attack is not offline detectable and that the adversary wins the subversion game. Theorems omitted due to lack of space and simplicity of proofs. The attack is online-detectable but not from a practical standpoint as discussed in Section C.1.

Leeching Attack. (§C.2) The subverted device attempts to exfiltrate valid puzzle solutions to the remote attacker. Such an attacker can then leverage these solutions to engage in a selfish mining [30] strategy, or to mount attacks against other networks. In *mining pools*, devices are asked to output low-difficulty

stateful if it requires $\overline{\mathsf{Eval}}$ to maintain state between invocations, and ***stateless*** otherwise. If the attack is stateless, then the state update function $\mathsf{st_{upd}}$ for the oracle running the subverted algorithm outputs the same state it takes as input.

[6] Offline detectability (5) implies online detectability (6) trivially as an adversary winning the offline detectability game can ignore the extra oracle access in the online detection game $\mathbf{Det}+^{\mathcal{D}}_{\mathsf{P},\bar{\mathsf{P}}}$

[7] While input-induced attacks have been studied in previous work [48,26], the process of triggering the subverted devices in the setting of blockchains has not received much scrutiny. We study this setting and come up with new ways to trigger subverted devices, sometimes even affecting the entropy on the blockchain to do so, as demonstrated in Section C.3.

[8] Alternatively it can just slow down the puzzle-solving process

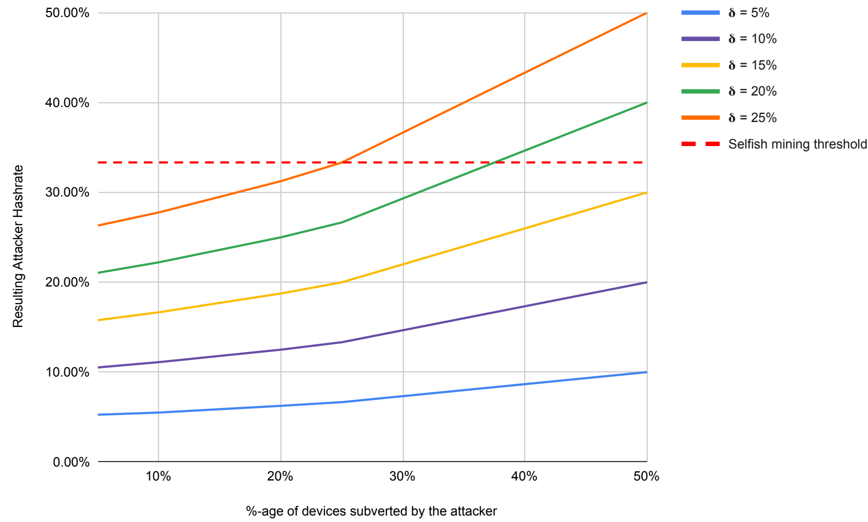


Fig. 4: Percentage (effective) increase in attacker hashrate plot as a function of the attacker's original hashrate (δ) and the percentage of subverted devices. This assumes that the subverted devices are unable to find a puzzle solution when under a load-shedding attack *in the direct input model.*

solutions periodically as "proof" that the device is contributing to the pool. These partial solutions allow for the creation of a subliminal channel that can be used to exfiltrate high-difficulty solutions when they are found. We devise a specialized subliminal channel exfiltration scheme using these partial solutions. Theorems 8, 9 prove that the loadshedding attack is not offline detectable and that the adversary wins the subversion game. The attack is online-detectable but not from a practical standpoint as discussed in §C.2.

Detecting active attacks. Classical ASA security definitions require that subversion must be undetectable even to a strong (i.e., polynomial-time) defender. The attacks we discuss in this work cannot fully achieve this goal, at least in the phase after an attack has been activated: each attack has a potentially detectable effect on the solution-rate of a target device. However, the fact that an attack is detectable in theory does not mean it can be detected *in practice.* Thus in this work we consider the requirements needed to detect real-world attacks on puzzle hardware. Surprisingly, we find that *for owners of individual devices* (and even large collections of subverted devices, see Figure 5) the time required to achieve high confidence that an attack is underway may exceed the working lifetime of the device(s) in question.

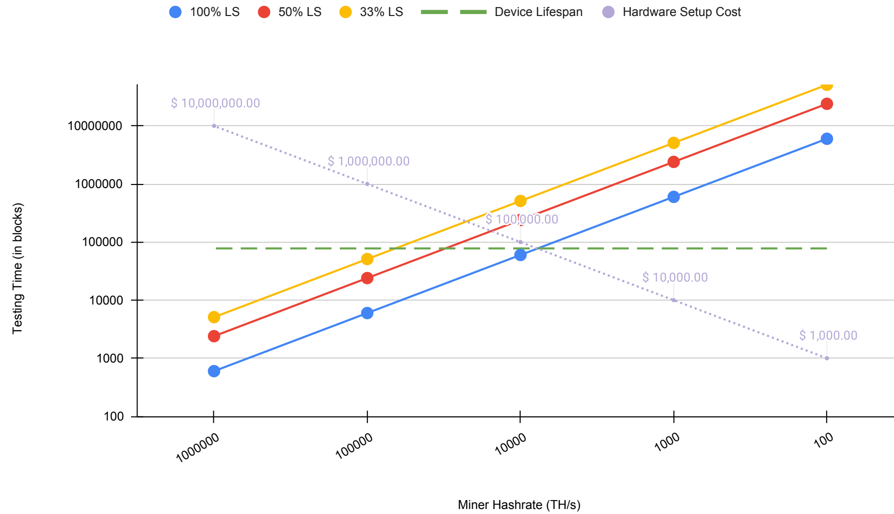


Fig. 5: Time taken (log scale) to statistically detect a (load-shedding) subverted device. Estimates based on Bitcoin mining 2022. Details available in Section 3.1.

3 Securing Puzzles against ASAs

We now know that the attacks can be dangerous and hard to detect. In this section, we look at the different ways to defend against such attacks. The two key directions we look at are: (i) testing regimes to figure out if a device is subverted and (ii) countermeasures against these attacks, which make it so that the attacker does not benefit from/can not trigger the subversion.

3.1 Testing

The best example of wide-scale use of puzzle evaluators in the real-world is proof-of-work mining hardware. Currently, there is no standardized testing against flaws, other than computing the number of operations per second. However, in light of our proposed attacks, owners of such devices might be interested in testing for subversions. However, every time period a mining device is not actually being used for mining, it is incurring an economic loss to its owner. More testing in this sense also benefits mining device manufacturers who also use their manufactured devices. Ironically, this can result in a paradox where distrust in the manufacturer benefits the attacker. The following techniques can be used to test for subversion:

Black-box statistical testing. A natural approach to detecting load-shedding and leeching attacks is to perform statistical tests to verify that the device

is producing solutions at the expected rate. However, this test can be time-consuming in networks (such as Bitcoin) where individual puzzle-solving devices find full-difficulty solutions only at long intervals. To evaluate the time and resources required to execute this testing strategy, we model this strategy for a hypothetical load-shedding attack against Bitcoin's proof-of-work in Figure 5. Suppose the hashrate of the miner is δ. We assume a 100% load-shedding attack (i.e., the victim's subverted device(s) never find a valid solution at the target difficulty.) We then determine the number of blocks b that the victim will process before they can determine with some chosen level of confidence that an attack may be underway. Here we pick a confidence level of 0.05, and the statistical test is: at what (smallest) value of b is it true that the probability of the miner with hashrate δ has not found a block is $< 0.05 = p$? When the miner is load-shed at lower values (e.g., 50% or 33% of solutions are discarded), this analysis must be adjusted accordingly. Taking as an example the case when the device is load-shed half of the time, the expected number of blocks found is $e = \frac{1}{2} \cdot \delta \cdot b$. Thus testing requires roughly number of blocks b such that $\sum_{x=0}^{e} \binom{b}{e} \cdot \delta^x \cdot (1-\delta)^{b-x} < 0.05 = p$. While we select conservative parameters for our test, our results in Figure 5 demonstrate a fundamental asymmetry between attacker and defender: while the attacker benefits from load shedding across the full network, individual victims may find it extremely costly and time-consuming to test their own devices. Even with an initial setup cost $> \$1m$, the test does not complete until a significant fraction of the device's lifespan. We assume (*i*) overall hashrate of the network is ≈200m TH/s and stays constant, (*ii*) it is profitable to use a mining device for 18 months ≈80k blocks [37], (*iii*) current cost of buying a device with 100 TH/s hashrate is around \$1000. A device with a capacity 100 TH/s currently consumes around 3 KW/h of power. Assuming a very conservative \$0.10 per KW/h of electricity, the cost of electricity spent is \$200,000 to have statistical confidence (p-value = 0.05) if the device is subverted. This is because regardless of your hashrate, the event used for testing the hypothesis (whether the device is subverted) happens once every block across the whole network. The cost is spent over 600 blocks (≈ 4 days) for a miner with a hashrate of 1m TH/s, and over 600,000 blocks (≈a decade) for a device with 100 TH/s. We plot the differences in the testing time based on the %-age of time the attacker load-sheds the subverted miner. While the lines are closer on the logarithmic scale, the testing time jumps up by 300% if the device is load-shed only half the time and ≈ 900% if its load-shed only a third of the time.) As the initial setup cost (dotted line) increases, the testing time decreases. Miners testing their own devices need to have a hardware setup cost much greater than \$100,000 to be able to complete the test before most of the device's lifespan is over. A possible solution to this problem is for many individual mining parties to share measurements and communicate their own test results to each other. For individual devices the attacks may *never* be detected: if the attacker decides to trigger the subversion shortly after the device is sold, ***it could take longer than the device's lifespan to complete such testing***.

Use existing puzzles with known solutions. In any proof-of-work cryptocurrency, solutions at the required difficulty level are found at least every few minutes. To test the mining devices, one can simply supply randomness ranges for the known solution in the past blocks, along with the other information (for Bitcoin prof-of-work this is the Merkle tree root of proposed block and the previous block hash as described in Appendix D). This strategy can detect obvious subversion. Unfortunately, this strategy doesn't work: an attacker who knows a set of past solutions at the time of device manufacture (or even subsequently, assuming an appropriate subliminal channel to the device) can configure the device to succeed on these known inputs[9], rendering this testing strategy ineffective.

Detecting leeching attacks. The leeching attack exfiltrates a valid solution from a subverted mining device, this raises the question of matching templates of accepted blocks to test against such attacks. Despite the theoretical possibility of comparing the block templates miners use with the blocks accepted into the blockchain, to the best of our knowledge, no existing software or systems facilitate this analysis for miners. To make such comparisons feasible, infrastructure to track submitted block templates would be required. It would need to be able to track and compare block templates globally. There is also the risk that information used for these comparisons could be misused. This could potentially expose miners to other types of attacks or invasions of privacy. Therefore, leeching attacks remain a significant and relevant threat in the context of blockchain consensus mechanisms as it stands today.

Devising testing-friendly puzzles. Many standard puzzle schemes require an unpredictable resource investment in order to solve a randomly chosen puzzle instance. As noted above, this can make testing challenging: for example, the Bitcoin puzzle (at current network difficulty levels) is sufficiently resource-intensive that a single device may never encounter a valid solution to a network-selected puzzle instance. An alternative approach to this problem is to devise puzzles that feature a property that we refer to as *test-friendliness*. A test-friendly puzzle is one that has a specialized algorithm Testgen that, on input some chosen test parameter ρ, produces a puzzle instance x with a known solution that can be found after approximately ρ resource-units by a solver executing a known strategy. Critically, such puzzle instances must be at least computationally *indistinguishable* from random puzzle instances that have similar resource requirements: this ensures that a subverted device cannot distinguish testing instances from "lucky" random instances generated by a network.[10] We describe a simple example of a test-friendly puzzle for Bitcoin in Appendix G.5.

[9] Note that a similar attacker strategy would prevent testing on lower difficulty solution from being effective for real-world attacks, the attacker triggers would only load-shed for difficulty levels above a set floor.

[10] Naturally such indistinguishability may not hold against *attacker-chosen* puzzle instances. We address defenses against this in the next section.

3.2 Unpredictable Puzzle Pre-processing

The unpredictability of the pre-processing that the puzzle construction supports captures how much control and flexibility the evaluator has to modify the structure of a puzzle once an instance is received. If this pre-processing is thorough enough, it can prevent an adversary from predicting the bits that will be input to the subverted device, which is all that is needed to prevent an adversary benefiting from an input-induced attack. Note that this is a generalization of defenses as proposed in [48] and their countermeasure is a special case of unpredictable pre-processing for hash functions only. And that cryptographic puzzles in general might allow for different techniques to achieve this, as shown by our protocol for masking Pietrzak VDFs.

Definition 2 (Unpredictable Pre-processing). *The pre-processing algorithm* Pre *has the unpredictability property if no PPT adversary* $\mathcal{A}$ *can distinguish the output of the algorithm from a random string with non-negligible probability, given only the unprocessed puzzle input and no knowledge of the randomness used to generate* aux.

$$\Pr\left[\begin{matrix} x \leftarrow \mathsf{Pre}(x', \mathsf{aux}), r \xleftarrow{\$} \mathcal{X}_{\mathsf{Pre}} \\ b' = b \end{matrix}\middle| \begin{matrix} b \xleftarrow{\$} \{0,1\}, \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta) \\ b' \leftarrow \mathcal{A}(1^\lambda, \Delta, \mathsf{pp}, x', b.x + (1-b).r) \end{matrix}\right] - \frac{1}{2} \leq \mathsf{negl}(\lambda)$$

Now, while the identity function is a validate candidate for the pre-processing algorithm Pre, it clearly does not have the unpredictability property. The Bitcoin proof-of-work (§D) puzzle is a good example of puzzles with unpredictable preprocessing.

Theorem 1. *In the direct input model, if the evaluator uses a puzzle pre-processing algorithm not satisfying the unpredictability property, all cryptographic puzzle schemes* $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ *are susceptible to subversion via any stateless load-shedding attacks where* $\frac{|\overline{\mathcal{X}}|}{2^\lambda} < \mathsf{negl}(\lambda)$.

Proof. This is a corollary of Theorem 7.

Theorem 2. *In the direct input model, if the evaluator uses a puzzle pre-processing algorithm satisfying the unpredictability property, then there exists no PPT attacker* $\mathcal{A}$ *which wins the subversion game* $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\bar{\mathsf{P}}}$ *with non-negligible advantage for any input-induced attack where* $\frac{|\overline{\mathcal{X}}|}{2^\lambda} < \mathsf{negl}(\lambda)$.

Theorem 3. *In the direct input model, all cryptographic puzzle schemes* $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ *utilizing a puzzle pre-processing algorithm satisfying the unpredictability property are secure against subversion via all input-induced attacks.*

Proofs for 2 and 3 are presented in Appendix F □

Protecting proof-of-work against ASAs. To ensure that the pooled mining protocols with pool operators allow unpredictable pre-processing of the proof-of-work puzzle input we suggest a new standardized protocol for pooled mining under pool operators (and not peer-to-peer mining). While the protocol described above is specifically for Bitcoin's proof-of-work, small modifications make it applicable to most proof-of-work cryptocurrencies. The miners participating in the pool then iterate over different values of the time and nonce fields as specified in the block hash algorithm (§D) to find proof-of-work solutions. he main change we propose is that the coinbase transaction should be modified by the miners. The coinbase transactions have extra fields for data and extraNonce which can be modified to provide up to 100 bytes [6] of extra space to iterate over for miners. This suggested change aims to allow miners the flexibility of picking the inputs to their mining devices. Modifications to the coinbase are not a new concept; the mining software can enforce such changes. We omit details here as similar changes have been suggested in the mining protocol by SlushPool [50]. We also recommend that all proof-of-work cryptocurrencies instruct users to only use pools with open-source mining software.

Theorem 4. *Proof-of-work pre-processing as specified above satisfies the unpredictability property as defined in definition 2.*

Proof sketch. Appears in Appendix F. □

Masking Pietrzak's VDFs. Verifiable delay functions (VDFs) are now being utilized by multiple blockchain protocols[11] [28,25]. We now discuss a method to mask the VDF input from a puzzle-solving device in case the input was adversarially biased to cause input-induced attacks. This is one way to ensure unpredictable puzzle pre-processing for VDFs. We demonstrate a masking protocol (Figure 1) using Pietrzak's VDF construction [43]. Our masking protocol changes the original scheme minimally and the extra computation incurred is also minimal, as demonstrated in Table 1. Our implementation is adapted from Bertron's work [17]. The numbers reflect computation times on a machine running Intel's i9-13900K CPU with a 32GB RAM. Note that these should be used for reference as when such a protocol is utilized at scale, it will be using specialized, blazing fast hardware [36] for the computation.

[11] A full description of Ethereum's Beacon Chain protocol is available in Appendix G.6.

$\|N\|$	2^T	Evaluation + Proving time (w/o Masking)	Pre-computation Cost ($\lceil\log(T)\rceil$ inverses)	Masking delay ($\lceil\log(T)\rceil$ multiplications)
512	2^8	1.1 ms	1.6×10^{-3} ms	2.4×10^{-5} ms
	2^{16}	21 ms	2.3×10^{-3} ms	3.1×10^{-5} ms
	2^{24}	1.6 s	3.1×10^{-3} ms	4.2×10^{-5} ms
	2^{32}	51 s	3.2×10^{-3} ms	4.4×10^{-5} ms
1024	2^8	3.7 ms	2.7×10^{-3} ms	4.0×10^{-5} ms
	2^{16}	79 ms	3.6×10^{-3} ms	5.1×10^{-5} ms
	2^{24}	5.9 s	5.3×10^{-3} ms	6.7×10^{-5} ms
	2^{32}	96 s	5.5×10^{-3} ms	6.9×10^{-5} ms
2048	2^8	17 ms	1.4×10^{-2} ms	1.9×10^{-4} ms
	2^{16}	586 ms	1.9×10^{-2} ms	2.6×10^{-4} ms
	2^{24}	15.4 s	3.3×10^{-2} ms	2.7×10^{-4} ms
	2^{32}	244 s	3.5×10^{-2} ms	3.0×10^{-4} ms

Table 1: Extra delay/computation required for Masked VDFs. For different lengths of N (RSA modulus) and exponentiations T, masking delay is a very small fraction of online delay, overhead $\frac{\log(T)}{T}$ decreases with increasing T.

Algorithm 1 Our Masked Halving Subprotocol ([43]'s subprotocol is available in Fig. 2)

1: **Prover masking** on input (N, x, T, y):
2: Picks a pre-evaluated b^{2^T} for a VDF instance b s.t.
3: $\alpha = \{b^{2^T}, b^{2^{T/2}}, \ldots, b^2\}$,
4: $\beta = \{(b^{2^T})^{-1}, (b^{2^{T/2}})^{-1}, \ldots, (b^2)^{-1}\}$
5: is precomputed
6: Sets $x_b = x.b$
7:
8: **VDF solver** $\mathcal{D}$ on input (N, x_b, T):
9: Computes $\gamma = \{x_b^{2^T}, x_b^{2^{T/2}}, \ldots, x_b^2\}$
10: Sends γ to **Prover**
11:
12: **Prover unmasks** by computing $y = y_b.(b^{2^T})^{-1} = \gamma[0].\beta[0]$
13: $\mathcal{P}, \mathcal{V}$ now have (N, x, T, y)
14:
15: At each halving step:
16: $\mathcal{P}$ computes:
17: $\mu = x_b^{2^{T/2}}.(b^{2^{T/2}})^{-1}$, $x' := x^r \cdot \mu$, $y' := \mu^r \cdot y$
18: $\mathcal{P}, \mathcal{V}$ interaction continues as specified

Pietrzak's VDF construction builds on the RSW [45] time-lock puzzle construction: this takes an element $x \in \mathbb{Z}_N^*$ and requires calculating $y = x^{2^T}$

Algorithm 2 Halving Subprotocol from [43]

1: On input (N, x, T, y)
2: Set $\mathcal{V}_{\mathsf{dec}} = \mathsf{nil}$
3: **while** $\mathcal{V}_{\mathsf{dec}} == \mathsf{nil}$ **do**
4:     **if** $T = 1$ and $y = x^{2^T}$ **then**
5:         $\mathcal{V}_{\mathsf{dec}} = 1$
6:     **else if** $T > 1$ **then**
7:         $\mathcal{P}$ computes $\mu = x^{2^{T/2}}$ and sends it to $\mathcal{V}$
8:         **if** $\mu \notin QR_N^+$ **then**
9:             $\mathcal{V}_{\mathsf{dec}} = 0$
10:             break
11:         **else**
12:             $\mathcal{V}$ samples a random $r \xleftarrow{\$} \mathbb{Z}_{2^\lambda}$ and sends it to $\mathcal{P}$
13:             **if** var is even **then**
14:                 $\mathcal{P}$, $\mathcal{V}$ output $(N, x', T/2, y')$
15:             **else**
16:                 $\mathcal{P}$, $\mathcal{V}$ output $(N, x', \frac{T+1}{2}, {y'}^2)$
17:             **end if**
18:         **end if**
19:         Set $T = T/2$
20:     **end if**
21: **end while**
22: Here $x' := x^r \cdot \mu \left(= x^{r + 2^{T/2}}\right)$ and $y' := \mu^r \cdot y \left(= x^{r \cdot 2^{T/2} + 2^T}\right)$

(mod N), where T specifies the puzzle difficulty. To convert this into a VDF, Pietrzak's work builds a protocol where a prover $\mathcal{P}$ convinces a verifier $\mathcal{V}$ that it solved an RSW puzzle. The VDF protocol utilizes the quadratic residue group $\left(QR_N^+, \circ\right)$ instead of $(\mathbb{Z}_N^*, \cdot)$.[12] The protocol proceeds as follows:

- The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ have an RSW puzzle (N, x, T) as common input along with the security parameter λ. Here $T \in \mathbb{N}$, $N = p \cdot q$ is the product of safe primes and the input $x \in QR_N^+$.
- The prover $\mathcal{P}$ computes T sequential squarings of the input x in the quadratic residue group QR_N^+ and sends $y = x^{2^T}$ to the verifier $\mathcal{V}$.
- The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ then iteratively engage in the "halving protocol" as described in Figure 2. This subprotocol starts with the common input (N, x, T, y) and the output is either $(N, x', \lceil T/2 \rceil, y')$ or the verifier outputs 0/1 and the protocol stops.

In the VDF hardware setting, the squarings and the halving protocol steps are computed in the hardware/computing device. The VDF computing device receives as input (N, x, T) and the randomness r for each round of the halving protocol. It outputs the puzzle solution y and the halving outputs for each round. We present our masked halving protocol in Figure 1 and show its correctness (§G.7). The goal is to ensure that no adversary who manufactured/subverted

[12] We refer readers to [43, §2.2] for a detailed discussion.

this device can feed in inputs and expect them to malfunction. Therefore, in the masked halving protocol there is another party the VDF computing device $\mathcal{D}$, other than the prover $\mathcal{P}$ and the $\mathcal{V}$. This party applies masking to ensure that each puzzle input to the VDF computing device is transformed unpredictably before the device sees it.

4 Conclusion

In this work, we examined the problem of attacking and defending puzzle-solving devices against subversion attacks. Beyond the attacks we propose and analyze, there are several real-world concerns relevant to puzzle-solving devices such as the limits of testing, some out-of-model attacks that are hard to prevent, and other carefully crafted subversion strategies. To that end, there is a huge need for such devices to be produced in facilities where the manufacturing process can be scrutinized. Creating processes to manufacture backdoor-resistant devices in that setting has already been studied in existing work [56,18]. However, today, there are billions of dollars worth of puzzle-solving devices already sold and used without such guarantees. We provide a discussion on all these concerns and consider stronger input-induced attacks and attackers in Appendix H. We hope this discussion serves as motivation for future work.

Acknowledgments

The first and second authors are supported in part by NSF under awards CNS-1653110 and CNS-1801479 and the Office of Naval Research under contract N00014-19-1-2292. The second author is also supported in part by DARPA under Contract No. HR001120C0084.

References

1. Equation Group: Questions and answers. Kaspersky Lab HQ (2015)
2. Hacker Infects Node.js Package to Steal from Bitcoin Wallets. Link. (2018)
3. Bitcoin mining producing tonnes of waste (2021), `https://www.bbc.com/news/technology-58572385`
4. Bitcoin's block hashing algorithm (2021), `https://en.bitcoin.it/wiki/Block_hashing_algorithm`
5. Bitcoin's difficulty based on network hashrate (2021), `https://en.bitcoin.it/wiki/Difficulty#What_network_hash_rate_results_in_a_given_difficulty.3F`
6. Bitcoin's transaction format (2021), `https://en.bitcoin.it/wiki/Transaction`
7. Bitmain (2021), `https://en.wikipedia.org/wiki/Bitmain`
8. Abusalah, H., Alwen, J., Cohen, B., Khilko, D., Pietrzak, K., Reyzin, L.: Beyond hellman's time-memory trade-offs with applications to proofs of space. In: ASIACRYPT 2017. pp. 357–379 (2017)

9. AntMiner: Antminer bitcoin mining device installation guide (2020), https://www.antminerdistribution.com/wp-content/uploads/2020/05/S19-Pro-manual.pdf
10. Ateniese, G., Magri, B., Venturi, D.: Subversion-resilient signature schemes. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS (2015)
11. Bellare, M., Jaeger, J., Kane, D.: Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks. In: ACM CCS 2015. pp. 1431–1440 (2015)
12. Bellare, M., Kane, D., Rogaway, P.: Big-key symmetric encryption: Resisting key exfiltration. In: Robshaw, M., Katz, J. (eds.) CRYPTO (2016)
13. Bellare, M., Paterson, K.G., Rogaway, P.: Security of symmetric encryption against mass surveillance. In: CRYPTO 2014. pp. 1–19 (2014)
14. Berndt, S., Liskiewicz, M.: Algorithm substitution attacks from a steganographic perspective. In: ACM CCS 2017. pp. 1649–1660 (2017)
15. Berndt, S., Wichelmann, J., Pott, C., Traving, T.H., Eisenbarth, T.: Asap: Algorithm substitution attacks on cryptographic protocols. In: ASIACCS (2022)
16. Bernstein, D.J., Lange, T., Niederhagen, R.: Dual EC: A standardized back door. In: Ryan, P.Y.A., Naccache, D., Quisquater, J. (eds.) The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday (2016)
17. Bertron, J.D.: Pietrzakvdf (2022), https://github.com/BqETH/PietrzakVDF, gitHub repository
18. Bhasin, S., Danger, J., Guilley, S., Ngo, X.T., Sauvage, L.: Hardware trojan horses in cryptographic IP cores pp. 15–29 (2013)
19. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: CRYPTO 2018 (2018)
20. Bonneau, J., Clark, J., Goldfeder, S.: On bitcoin as a public randomness source. IACR Cryptol. ePrint Arch. p. 1015 (2015)
21. BTC.com: Blocks proposed by various bitcoin pools 2021-22 (2022), https://btc.com/stats/pool?pool_mode=year
22. Checkoway, S., Maskiewicz, J., Garman, C., Fried, J., Cohney, S., Green, M., Heninger, N., Weinmann, R., Rescorla, E., Shacham, H.: A systematic analysis of the juniper dual EC incident. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS (2016)
23. Chen, R., Huang, X., Yung, M.: Subvert KEM to break DEM: practical algorithm-substitution attacks on public-key encryption. In: ASIACRYPT 2020. pp. 98–128 (2020)
24. Chow, S.S.M., Russell, A., Tang, Q., Yung, M., Zhao, Y., Zhou, H.: Let a non-barking watchdog bite: Cliptographic signatures with an offline watchdog. In: Lin, D., Sako, K. (eds.) Public-Key Cryptography - PKC (2019)
25. Cohen, B., Pietrzak, K.: The chia network blockchain (2019), https://www.chia.net/greenpaper/
26. Degabriele, J.P., Farshim, P., Poettering, B.: A more cautious approach to security against mass surveillance. In: Leander, G. (ed.) Fast Software Encryption FSE 2015. Lecture Notes in Computer Science, Springer (2015)
27. Dodis, Y., Gennaro, R., Håstad, J., Krawczyk, H., Rabin, T.: Randomness extraction and key derivation using the cbc, cascade and HMAC modes. In: CRYPTO 2004. pp. 494–510 (2004)
28. Drake, J.: Minimal vdf randomness beacon. In: Ethereum Research Forum (2018), https://ethresear.ch/t/minimal-vdf-randomness-beacon/3566
29. Dziembowski, S., Faust, S., Kolmogorov, V., Pietrzak, K.: Proofs of space. In: CRYPTO 15. pp. 585–605 (2015)

30. Eyal, I., Sirer, E.G.: Majority is not enough: Bitcoin mining is vulnerable. In: Financial Cryptography and Data Security FC 2014 (2014)
31. Fischlin, M., Mazaheri, S.: Self-guarding cryptographic protocols against algorithm substitution attacks. In: IEEE CSF 2018. pp. 76–90 (2018)
32. Groza, B., Warinschi, B.: Cryptographic puzzles and dos resilience, revisited. Des. Codes Cryptogr. **73**(1), 177–207 (2014)
33. Hodges, P., Stebila, D.: Algorithm substitution attacks: State reset detection and asymmetric modifications. IACR Trans. Symmetric Cryptol. **2021**(2), 389–422 (2021)
34. Juels, A., Brainard, J.G.: Client puzzles: A cryptographic countermeasure against connection depletion attacks. In: NDSS (1999)
35. Khovratovich, D., Maller, M., Tiwari, P.R.: Minroot: Candidate sequential function for ethereum VDF. p. 1626 (2022)
36. Kim, C.: Ethereum foundation and others weigh $15 million bid to build randomness tech. In: CoinDesk (2019)
37. Köhler, S., Pizzol, M.: Life cycle assessment of bitcoin mining - acs publications, https://pubs.acs.org/doi/10.1021/acs.est.9b05687
38. Labs, M.M.: 51% attacks on cryptocurrencies (2020), https://dci.mit.edu/51-attacks
39. Luu, L., Velner, Y., Teutsch, J., Saxena, P.: SmartPool: Practical decentralized pooled mining. In: 26th USENIX Security Symposium (USENIX Security 17) (2017)
40. Miller, A., Kosba, A.E., Katz, J., Shi, E.: Nonoutsourceable scratch-off puzzles to discourage bitcoin mining coalitions. In: ACM CCS (2015)
41. Mining, Z.A.: Explanation of the number of antminer hardware errors (2022), Link.
42. Pierrot, C., Wesolowski, B.: Malleability of the blockchain's entropy. Cryptography and Communications **10**(1), 211–233 (2018)
43. Pietrzak, K.: Simple verifiable delay functions. In: ITCS 2019. pp. 60:1–60:15 (2019)
44. Redman, J.: Mining report highlights china's asic manufacturing improvements and dominance (2020), Link to article.
45. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto (1996)
46. Russell, A., Tang, Q., Yung, M., Zhou, H.: Generic semantic security against a kleptographic adversary. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM SIGSAC CCS 2017
47. Russell, A., Tang, Q., Yung, M., Zhou, H.: Cliptography: Clipping the power of kleptographic attacks. In: ASIACRYPT. Lecture Notes in Computer Science (2016)
48. Russell, A., Tang, Q., Yung, M., Zhou, H.S.: Correcting subverted random oracles. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018 (2018)
49. SlushPool: Bitcoin mining pools: Luck, shares, and estimated hashrate explained (2021), Link.
50. SlushPool: Stratum v1 docs: Mining protocol (2022), https://braiins.com/stratum-v1/docs
51. Stebila, D., Kuppusamy, L., Rangasamy, J., Boyd, C., Nieto, J.M.G.: Stronger difficulty notions for client puzzles and denial-of-service-resistant protocols. In: CT-RSA 2011. pp. 284–301 (2011)
52. Stefanko, L.: Crypto malware in patched wallets targeting android and ios devices. Link. (2022)
53. Taylor, A.: Watch out for the 'rug pull' crypto scam that's tricking investors out of millions (2022), https://fortune.com/2022/03/02/crypto-scam-rug-pull-what-is-it/

54. Têtu, J., Trudeau, L., Beirendonck, M.V., Balatsoukas-Stimming, A., Giard, P.: A standalone fpga-based miner for lyra2rev2 cryptocurrencies. IEEE Trans. Circuits Syst. I Fundam. Theory Appl. (2020)
55. Tromp, J.: Cuckoo cycle: A memory bound graph-theoretic proof-of-work. In: Brenner, M., Christin, N., Johnson, B., Rohloff, K. (eds.) BITCOIN workshop (FC) (2015)
56. Wahby, R.S., Howald, M., Garg, S., Shelat, A., Walfish, M.: Verifiable asics. In: IEEE Symposium on Security and Privacy, SP. IEEE Computer Society (2016)
57. Wesolowski, B.: Efficient verifiable delay functions. In: EUROCRYPT 2019 (2019)
58. Williams-Grut, O.: Chinese bitcoin mining giant bitmain had revenues of $2.8 billion in the first half of the year (2018), Link to article.
59. Wong, J.I.: Research: Hackers could install backdoor in bitcoin cold storage (2015), Link to the article.
60. Wright, T.: Samson mow: Bitmain s17/t17 antminer has high failure rate (2020), Link.
61. Young, A.L., Yung, M.: Kleptography: Using cryptography against cryptography. In: Fumy, W. (ed.) EUROCRYPT 1997 (1997)

A Cryptographic Puzzles

To provide background for the attacks in this work, we require a general definition of cryptographic puzzles that capture the various cryptographic primitives that are used as puzzles. Our definitions simplify and extend a previous definition by Groza and Warinschi [32]: they study cryptographic puzzles with different difficulty requirements for puzzle solving from the perspective of puzzle generation. For a detailed discussion on puzzle difficulty bounds we refer readers to their work.[13] Our goal is to generalize and simplify the cryptographic puzzle definitions to capture a variety of common puzzles, including client puzzles [51,34], proof of work, verifiable delay functions (VDFs) and proof of space. In all applications, these puzzles require some amount of resource to solve. The nature of this resource varies between puzzles: total computation cycles in proof-of-work, sequential-time for VDFs, and storage space for proof-of-space puzzles. Our definition is designed to generalize over various types of resources as well.

Definition 3 (Cryptographic Puzzles). *A cryptographic puzzle is a tuple* $(\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ *comprising the possibly probabilistic algorithms defined in Section 2.*

As in previous works, we omit an explicit puzzle-sampling algorithm and specify that puzzles are sampled uniformly from the domain $\mathcal{X}_{\mathsf{Pre}}$. *We also slightly modify previous definitions to* split *the puzzle evaluation algorithm into a new (optional)*

[13] Groza and Warinschi's definitions consider other properties of the puzzle from the perspective of the puzzle evaluator such as *optimality*, which tightly bounds the success probability of puzzle solving and *fairness*, which bounds the probability of an evaluator finding a puzzle solution after some number of computational steps. These puzzle properties are extremely meaningful, but not directly relevant to our work.

pre-processing *algorithm* Pre *and an evaluation algorithm* Eval*. This modification captures previous definitions, and will be used in several of our later defenses.*

Cryptographic puzzles must satisfy an intuitive definition of correctness (definition 8) and soundness (definition 9), which respectively ensure that (1) any correct solution can be successfully verified by the Verify algorithm, and (2) an adversary cannot persuade a verifier to accept an incorrect puzzle solution. Additionally, we formalize a resource requirement definition (10) which allows us to specify the minimum or average resource investment a puzzle solver needs to find a solution to a puzzle instance. For space reasons we present the full definitions in the Appendix.
To make the above discussion more concrete, we now consider several real-world cryptographic primitives that instantiate the cryptographic puzzle formalism:
Nakamoto Proof-of-Work. The Nakamoto proof-of-work (PoW) is a costly, time-consuming computation that miners conduct to select the node that produces the next consensus block. While several currencies employ proof-of-work, we will focus on the Bitcoin proof-of-work (Section D) as a specific example. At the time of writing, Bitcoin provides a 6.25 BTC *block reward* to the miner who solves this puzzle (> $ 100k.) The Bitcoin proof of work can be formalized as a cryptographic puzzle as described in Appendix 11.
Verifiable Delay Functions. A verifiable delay function (VDF) [19,43,57,35] is a specialized puzzle where computing the solution on any instance requires running a fixed number of *sequential* steps: at the same time, the output of the puzzle can be efficiently verified. VDF definitions and security requirements are available in Appendix 11. Multiple applications of VDFs [28] have sparked a coalition called "VDF Alliance" funded by the Ethereum Foundation, Protocol Labs and others. VDFs can also be formalized in our framework as demonstrated in Appendix 15.
Proof of Space. Proof of Space (PoSpace) [29,8,25] is a puzzle that was developed as an alternative to Bitcoin's proof of work. Unlike standard PoW algorithms, proof of space puzzles use disk space as the puzzle resource rather than computation. PoSpace definitions and security requirements are presented in Appendix 11. PoSpace is also formalized as a cryptographic puzzle in Appendix 17.

B Security Definitions and Games

Definition 4 (Subversion resistance). *A cryptographic puzzle scheme* $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ *is subversion resistant against an algorithm-substitution attack if for the subverted puzzle algorithms* $\overline{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$ *all subverting adversaries/attacker* $\mathcal{A}$ *have a negligible advantage in the subversion game* $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\bar{\mathsf{P}}}$*, i.e.,* $\forall(\lambda, \Delta, \mathcal{A})$*:*

$$\mathbf{Adv}^{\mathbf{Sub}}_{\mathcal{A}}(1^{\lambda}, \Delta) \leq \mathsf{negl}(\lambda)$$

Definition 5 (Offline Detectability). *An algorithm-substitution attack against a cryptographic puzzle scheme* $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$*, where the subverted*

puzzle algorithms are represented by $\bar{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$*, is considered detectable if there exists a detection adversary with non-negligible advantage in the detection game* $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\bar{\mathsf{P}}}$*, i.e.,* $\forall(\lambda, \Delta), \exists\mathcal{D}$ *such that:*

$$\mathbf{Adv}^{\mathbf{Det}}_{\mathcal{D}}(1^{\lambda}, \Delta) > \mathsf{negl}(\lambda)$$

Definition 6 (Online Detectability). *An algorithm-substitution attack against a cryptographic puzzle scheme* $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$*, where the subverted puzzle algorithms are represented by* $\bar{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$*, is considered detectable if there exists a detection adversary with non-negligible advantage in the online detection game* $\mathbf{Det}+^{\mathcal{D}}_{\mathsf{P},\bar{\mathsf{P}}}$*, i.e.,* $\forall(\lambda, \Delta), \exists\mathcal{D}$ *such that for all negligible functions* $\mathsf{negl}(\cdot)$*:*[14]

$$\mathbf{Adv}^{\mathbf{Det}+}_{\mathcal{D}}(1^{\lambda}, \Delta) > \mathsf{negl}(\lambda)$$

[14] Note that if an attack is only online detectable then it is stronger as it requires a stronger detector

Game Definition 1: the detection game $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\bar{\mathsf{P}}}$ (offline), as referenced in Section 2.1

$\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\bar{\mathsf{P}}}$**:**

- **Setup.** In the setup phase, the challenger samples a bit $b \xleftarrow{\$} \{0,1\}$. If $b = 0$, the challenger runs the unsubverted puzzle algorithms $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ and sends the output of $\mathsf{Setup}(1^\lambda, \Delta)$ to the adversary $\mathcal{D}$. Otherwise, it runs the subverted puzzle algorithms $\bar{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$ and sends the output of $\overline{\mathsf{Setup}}(1^\lambda, \Delta)$ to $\mathcal{D}$.
- **Query.** In the query phase, the adversary gets access to one of the puzzle evaluation oracles $\mathsf{O}_{\mathsf{Eval}}$ or $\mathsf{O}_{\overline{\mathsf{Eval}}}$ depending on the challenger's sampled bit b being 0 or 1. The adversary can query the oracle polynomially many times in the security parameter. These oracles are defined as follows:
  - $\mathsf{O}_{\mathsf{Eval}}(\mathsf{pp}, \cdot, \cdot)$ takes as input an element $x \in \mathcal{X}$ and the corresponding pre-processing auxiliary input aux. It computes and outputs $y \leftarrow \mathsf{Eval}(x, \mathsf{aux})$ and a counter cnt indicating the time spent in computing the output.
  - $\mathsf{O}_{\overline{\mathsf{Eval}}}(\mathsf{pp}, \mathsf{state}, \cdot, \cdot)$ takes as input an element $x \in \mathcal{X}$, the corresponding pre-processing auxiliary input aux and also maintains an internal state state. It computes and outputs $\bar{y} \leftarrow \overline{\mathsf{Eval}}(x, \mathsf{aux})$ and a counter cnt indicating the time spent in computing the output. The state update function, sets new state as $\mathsf{state}' \leftarrow \mathsf{st}_{\mathsf{upd}}(\mathsf{state}, x)$.
- **Guess.** In this phase, the adversary outputs its guess b' for b and wins the detection game if $b' = b$. We say that the game outputs 1 if the adversary wins and 0 otherwise. The advantage of an adversary $\mathcal{D}$ is defined as
  $\mathbf{Adv}^{\mathbf{Det}}_{\mathcal{D}}(1^\lambda, \Delta) = \Pr[b' = b] - 1/2$.

Game Definition 2: the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\bar{\mathsf{P}}}$, as referenced in Section 2.1

$\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\bar{\mathsf{P}}}$:

- **Setup.** In the setup phase, the challenger samples a bit $b \xleftarrow{\$} \{0,1\}$. If $b = 0$, the challenger runs the unsubverted puzzle algorithms $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ and sets $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta)$ to the adversary $\mathcal{A}$. Otherwise, it runs the subverted puzzle algorithms $\bar{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$ and sets $\mathsf{pp} \leftarrow \overline{\mathsf{Setup}}(1^\lambda, \Delta)$. It sends $(\mathsf{pp}, \bar{\mathsf{P}})$ to the adversary $\mathcal{A}$.
- **Query.** In the query phase, the adversary $\mathcal{A}$ gets access to one of the puzzle evaluation oracles $\mathsf{O}_{\mathsf{Eval}}$ or $\mathsf{O}_{\overline{\mathsf{Eval}}}$ depending on the challenger's sampled bit b being 0 or 1. The adversary can query the oracle polynomially many times in the security parameter. These oracles are exactly the same in the detection game $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\bar{\mathsf{P}}}$.
- **Guess.** In this phase, the adversary outputs its guess b' for b and wins the detection game if $b' = b$. We say that the game outputs 1 if the adversary wins and 0 otherwise.
  The advantage of an adversary $\mathcal{D}$ is defined as $\mathbf{Adv}^{\mathbf{Sub}}_{\mathcal{A}}(1^\lambda, \Delta) = \Pr[b' = b] - 1/2$.

Game Definition 3: the detection game $\mathbf{Det}{+}^{\mathcal{D}}_{\mathsf{P},\bar{\mathsf{P}}}$ (online), as referenced in Section 2.1

$\mathbf{Det}{+}^{\mathcal{D}}_{\mathsf{P},\bar{\mathsf{P}}}$:

In this variant of the detection game, the adversary also has black-box access to an attacker. The attacker is an adversary in the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\bar{\mathsf{P}}}$ that wins with some non-negligible probability. We describe the attack oracle as follows:

- $\mathsf{O}^{l}_{\mathsf{Attack}}(\mathsf{pp}, \cdot, \cdot)$ takes as input the public parameters, a symbol s indicating the start of an attack instance and a tuple $t = (x, y, \mathsf{cnt})$ indicating device's output on the latest input instance. The symbol $\mathsf{s} = 1$ indicates the start of a fresh oracle query, this can be accompanied by $t = (\perp, \perp, \perp)$ to indicate a clear start of an attack instance. All following queries have inputs of type either (i) $\mathsf{s} = 0$ (indicating continuation) and $t = (x, y, \mathsf{cnt})$ or (ii) $\mathsf{s} = 1$ (indicating oracle reset) and $t = (\perp, \perp, \perp)$. The oracle outputs a puzzle input instance $x \in \mathcal{X}$ and the corresponding pre-processing auxiliary input aux from a subverting adversary/ attacker that wins the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\bar{\mathsf{P}}}$. The oracle is set to allow resetting up to l times.

C Attacks

We now describe our general attack strategies, followed by the challenges in the two different threat models/attack settings as described in Section 2. While we emphasize attacks that are input-induced attacks in the black-box setting, even if the devices are sold in the subverted state (no trigger needed to subvert), the attack strategies would still work well. Such a setting is reflected in scenarios where a puzzle solver buys some puzzle-solving hardware from an untrusted party. A simple example of an input-induced attack is to have a set of hard-coded triggers in the puzzle evaluation hardware/software on which the evaluation fails/does not proceed as expected. However, in this case, the set of such triggers will need to be negligibly small compared to the puzzle input domain in order to avoid detection.

Trigger strategy. At manufacturing time, the untrusted party must select a set of trigger inputs $\overline{\mathcal{X}}$, where $\overline{\mathcal{X}} \subset \mathcal{X}$ such that if $x \in \overline{\mathcal{X}}$ then the evaluation algorithm misbehaves, as specified by the subversion. Therefore, in both our attack settings for a cryptographic puzzle $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ the subverted puzzle algorithms are of the form $\overline{\mathsf{P}} = (\mathsf{Setup}, \mathsf{Pre}, \overline{\mathsf{Eval}}, \mathsf{Verify})$. Due to the black-box nature of the attack setting, any detector can only detect the subversion through oracle access.

$\overline{\mathsf{P}}.\overline{\mathsf{Eval}}(\mathsf{pp}, x, \overline{\mathcal{X}})$: *The subverted puzzle evaluation algorithm takes as input the public parameters* pp*, an input* $x \in \mathcal{X}$*, and a set* $\overline{\mathcal{X}} \subset \mathcal{X}$*. If* $x \in \mathcal{X} \setminus \overline{\mathcal{X}}$ *it outputs a puzzle solution* y*. Otherwise, the evaluation algorithm fails to produce an output.*

C.1 Load-Shedding Attack

The goal of a load-shedding attack is to slow down the solution rate of the puzzle-solving algorithm. Depending on the application, the attacker could make it so that the subverted device just discards the puzzle solution (such as the proof-of-work puzzle solution) or reduce puzzle-solving ability slightly (would cause extra delay for finding VDF solutions). If an attacker can cause this to happen across a large enough fraction of a network's device capacity, the overall network solution rate reduces and unsubverted devices will comprise a higher percentage of the network's capacity. Furthermore, load-shedding can be combined with strategies such as selfish mining [30, Appendix D] to enable efficient consensus attacks on certain networks even when the attacker directly possesses a relatively small fraction of the hashrate (see Figure 4). If this happens only occasionally, the attacker may be able to repeat this attack undetected for a long period of time. A similar attack strategy is proposed for hash functions in [48]: there the hash function is different from an implementation on certain inputs. We decisively settle the debate on whether this is a theoretical attack idea: ***it is not***. We generalize this attack strategy for all cryptographic puzzles, and our analysis in Figure 4 and Figure 3 shows that certain entities could have been benefitting from selling subverted devices. Later sections and Figure 5 show that it is hard to determine whether a particular device is subverted even when testing extensively.

We first prove some general results on load-shedding attacks where the size of the adversarially hard-coded triggers is negligible in the security parameter. This models the setting where the detector $\mathcal{D}$ is testing its hardware to ensure proper performance. Since the set of bad inputs is negligibly small, the probability of detection is negligibly low. However, when the attacker plays the subversion game, it can easily detect whether the challenger is operating a subverted evaluation algorithm. This is because the attacker knows the set of hard-coded bad inputs.[15]

Theorem 5. *For all load-shedding attacks where $\frac{|\overline{\mathcal{X}}|}{|\mathcal{X}|} < \mathsf{negl}(\lambda)$, there exists no PPT detector $\mathcal{D}$ which wins the offline detection game $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$ with non-negligible advantage.*

Proof. The set of subverted inputs $\overline{\mathcal{X}}$ is a random subset of the set of inputs $\mathcal{X}$. Given that $\frac{|\overline{\mathcal{X}}|}{|\mathcal{X}|} < \mathsf{negl}(\lambda)$, for a PPT detector $\mathcal{D}$, the advantage of winning the detection game $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$ is same as the probability of querying the evaluation oracle on at least one input $x \in \overline{\mathcal{X}}$. Let event E be the query instance when $\mathcal{D}$ queries the evaluation oracle on an input $x \in \overline{\mathcal{X}}$. Let $\Pr[\overline{E}] = 1 - \Pr[E]$ be the probability of event $\overline{E}$ (complement of event E), i.e. , for query x of $\mathcal{D}$, $x \notin \overline{\mathcal{X}}$. Let q be the number of queries that $\mathcal{D}$ asks. Then,

$$\mathbf{Adv}^{\mathbf{Det}}_{\mathcal{D}}(1^\lambda, \Delta) = 1 - \left(\Pr[\overline{E}]\right)^q$$

$$= 1 - \left(\Pr[x \in \mathcal{X} \setminus \overline{\mathcal{X}}]\right)^q = 1 - (1 - \mathsf{negl}(\lambda))^q$$
$$\approx 1 - (1 - q.\mathsf{negl}(\lambda)) = q.\mathsf{negl}(\lambda)$$

Since, q is the number of queries $\mathcal{D}$ asks, $q = \mathsf{poly}(\lambda)$. Therefore,

$$\mathbf{Adv}^{\mathbf{Det}}_{\mathcal{D}}(1^\lambda, \Delta) = q.\mathsf{negl}(\lambda) = \mathsf{negl}'(\lambda)$$

Theorem 6. *For all load-shedding attacks, there exists a PPT attacker $\mathcal{A}$ which wins the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\overline{\mathsf{P}}}$ with non-negligible advantage.*

Proof. The attacker $\mathcal{A}$ leverages its knowledge of the set of subverted inputs $\overline{\mathcal{X}}$. Upon receiving the public parameters pp and the subverted algorithms $\overline{\mathsf{P}} = (\overline{\mathsf{Setup}}, \overline{\mathsf{Pre}}, \overline{\mathsf{Eval}}, \overline{\mathsf{Verify}})$, in the query phase, the attacker first queries on inputs $x \in \mathcal{X} \setminus \overline{\mathcal{X}}$. After this, the attacker picks $x' \in \overline{\mathcal{X}}$ as a query. Using the expected response time from querying on the unsubverted set of inputs, the attacker decides whether its querying the subverted oracle or not based on the (timed) query response received for the subverted input x'. This is because the unsubverted oracle would respond with a correct solution in the expected time.

[15] We assume for theorems 5-7 that the puzzle evaluator does not utilize the puzzle pre-processing algorithm and feeds puzzle inputs without pre-processing to the evaluation hardware/software

Theorem 7 (offline-security). *All cryptographic puzzle schemes* $\mathsf{P} = (\mathsf{Setup}, \mathsf{Eval}, \mathsf{Verify})$ *are susceptible to subversion via any load-shedding attacks where* $\frac{|\overline{\mathcal{X}}|}{2^\lambda} < \mathsf{negl}(\lambda)$.

Proof. This follows from Theorem 5 and Theorem 6 under definition 1. □

Online-security against load-shedding. We discuss this issue separately as the online-security of a puzzle-solving device against a load-shedding attack depends on the real-world use case. For example, for proof-of-work, most mining devices operate within an acceptable error range of puzzle-solving ability. If the drop in performance is within the range, the device owner can possibly blame it on the physical conditions (temperature etc.) in which the device is hosted. Depending on how often the adversary mounts a load-shedding attack, the drop in puzzle-solving ability can still be in the acceptable range and it's never detected. However, a load-shedding attack would be online detectable in the sense of definition 6 as in an ideal world, the device has a given, set puzzle-solving ability, and deviation from that would be evidence for the detector that the device is subverted.

C.2 Leeching Attack

The goal of the adversary with the leeching attack is to compromise the puzzle evaluation hardware such that (when triggered) it exfiltrates puzzle solutions to the attacker.[16] We use proof-of-work mining pools as a high-impact example. The subverted evaluation algorithm tries to exfiltrate the first solution it finds to the puzzle without outputting it and continues to find the second solution, which it then outputs. The exfiltration process is explained in full detail in Appendix G.2, it uses lower-difficulty solutions[17] as the exfiltration channel. Such exfiltration is not unrealistic because the puzzle evaluation hardware is an online device, ex. bitcoin mining rigs [9, Page 5]. Clearly, in the example of selfish mining such an attack is very beneficial. *Now, the attacker does not need to own a third of the hashrate but instead just needs to subvert enough devices to reach the attack threshold.*

We again proceed by first proving some **general results** on leeching attacks where the size of the adversarially hard-coded bad inputs is negligible in the security parameter.

Theorem 8. *For all leeching attacks where* $\frac{|\overline{\mathcal{X}}|}{2^\lambda} < \mathsf{negl}(\lambda)$, *there exists no PPT detector* $\mathcal{D}$ *which wins the offline detection game* $\mathbf{Det}^{\mathcal{D}}_{\mathsf{P},\overline{\mathsf{P}}}$ *with non-negligible advantage.*

Proof sketch. Proof arguments are exactly the same as the proof for Theorem 5.

[16] This can be thought of as a variant of load-shedding but with exfiltration

[17] Lower-difficulty solutions in proof-of-work are used as a metric to determine how much computational power a participant is contributing to the mining pool. This value is then utilized for mining reward distribution.

Theorem 9. *For all leeching attacks, there exists a PPT attacker $\mathcal{A}$ which wins the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\bar{\mathsf{P}}}$ with non-negligible advantage.*

Proof sketch. Proof arguments are exactly the same as the proof for Theorem 6.

Theorem 10 (offline-security). *If there exists a secure exfiltration channel* E*, all cryptographic puzzle schemes* $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ *are susceptible to subversion via any leeching attacks where* $\frac{|\overline{\mathcal{X}}|}{2^\lambda} < \mathsf{negl}(\lambda)$.

Proof sketch. Since this attack and exflitration becomes relevant in the online phase, in the offline phase, security depends on testing the device on a trigger element $x' \in \overline{\mathcal{X}}$. This, along with the fact that any subverting adversary can win the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\bar{\mathsf{P}}}$ (Theorem 9), viewed under definition 1, completes our proof. □

Online-security against Leeching. Similar to online-security of load-shedding, we discuss this issue separately as the online-security of a puzzle-solving device against this attack depends on the real-world use-case. In proof-of-work, most mining devices have been exhibiting "hardware errors"(Appendix G.1), these are inputs on which the device hardware does not produce an output. From the resources compiled in Appendix G.1, we can see that accumulating a few hundred errors after running a mining device for a few days is considered acceptable by the community. Mining device owners across the world have discussed this issue and the general consensus is that this is due to pushing the devices to the absolute edge of performance. *This is exactly how things would appear to a device owner if a leeching attack was underway.*

C.3 Analyzing attack strategies

We now discuss challenges in the different threat models.

Direct Input Model. In the direct input model, an attacker can mount both load-shedding and leeching attacks. The attacker directly sends puzzle inputs to the subverted device. If there is a way to exfiltrate puzzle solutions (for example, in the form of lower-level difficulty solutions as used in proof-of-work to demonstrate the contribution of efforts), then these allow for leeching attacks as well. The attacker has complete control over what is input into the device. [18]

Blockchain Input Model. In the blockchain input model, an attacker can only mount a load-shedding attack. The attacker is leveraging the fact that the mining device reads from the blockchain. Therefore, the adversary's influence on

[18] This allows the attacker to trigger the device whenever it chooses to do so. An attacker can combine many clever strategies with load-shedding to go undetectable. For example, the attacker could have subverted the device so that on observing a trigger, the device's puzzle-solving ability only declines for the next hour for example. This gives the attacker the fine-grained ability to trigger devices temporarily,the idea is to go undetected but continually benefit from the subversion.

the blockchain's state determines the strength of any such attack. The communication channel here only goes from the attacker to the subverted device via the blockchain. Hence, no exfiltration of puzzle solutions and, consequently, no leeching attacks are possible. Assuming any such adversary will own some fraction of the puzzle-solving resource in the blockchain's network, it's attempt to bias the blockchain is captured by the $\delta^{\mathcal{A}}_{\mathsf{bias}}$ game played between a set of honest miners/puzzle solvers and an adversary $\mathcal{A}$. The goal of the adversary is to bias the probability distribution of the extract of a *decisive block*, which is a future block of interest to the adversary. The adversary holds influence over a δ fraction of the blockchain network's puzzle solving resource. The adversary wins if the extract x of the decisive block B, $\mathsf{Ext}(B) = x$ falls in a subset favorable to the adversary. This model is inspired by the work of Pierrot and Wesolowski [42], which models an adversary who biases a blockchain's entropy. The assumption that each new block has some min-entropy follows from the work of Bonneau, Clark and Goldfeder [20]. We describe the models in these related works in Appendix G.3.

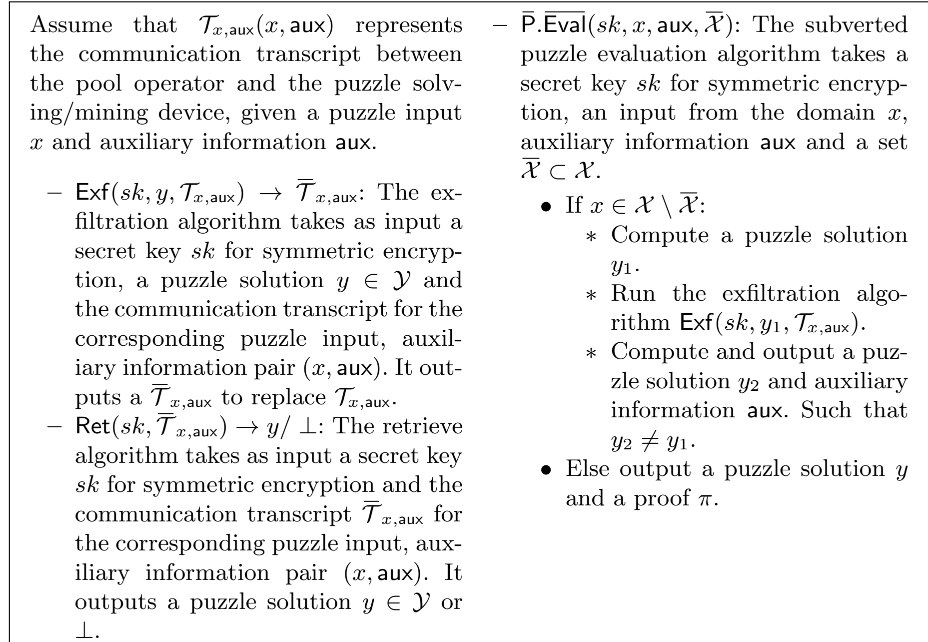
Definition 7 (δ-bias Ability). *An adversary $\mathcal{A}$ has δ-bias ability if for $s_{\mathcal{A}}$ being the number of puzzle solutions per second the adversary can calculate and s being the number of puzzle solutions per second all miners combined (including $\mathcal{A}$) can calculate,*

$$\delta = \frac{s_{\mathcal{A}}}{s}$$

This is a lower bound of bias considering the hashrate controlled by the attacker. The attacker might collude with or bribe other parties to increase their bias ability. Building over the analysis from [42]: we determine that the load-shedding attack does not perform well for the adversary in the blockchain input model. A detailed analysis is available in Section G.4. We now discuss a stateful variant of this attack that considerably improves efficacy.

Stateful triggers. In the stateful variant of each attack, the attacker can spread the trigger over several messages and cause the attack to continue for a pre-specified period of time. This is a crucial attack variant due to the fact that an adversary mounting a stateful attack needs much lesser influence over the blockchain's state. For example, a simple strategy could fix the low-order bit in the block header and subvert the puzzle-solving device so that if it sees n consecutive blocks such that the low-order bits form a particular n-bit string (some $x \in \overline{\mathcal{X}}$) then it fails to compute the puzzle solution completely or becomes much slower. The main change in strategy is that the attacker can now feed in triggers over a longer period. Similar to previous attacks, the more influence the attacker holds over the blockchain's state, the stronger the attack. More formally, the adversary's aim is to bias the blockchain's state over n blocks, such that for some encoding Enc, $\mathsf{Enc}(B_n) = 1$, where B_n represents the state of the blockchain over the last n blocks. The mathematical analysis for the adversary's success probability is available in Appendix G.4. We plot the values of n, the (Reed-Solomon) encoding parameters, and the probability of success

next to different values of the adversarial resource fraction δ in Table 2. Note that such an attack can not be detected by any detection adversary (Theorem 5), and any attacker can figure out if the puzzle-solving device is running a subverted variant of the puzzle evaluation algorithm by simply checking if the encoding of the blockchain's state in the last n blocks, $\mathsf{Enc}(B_n)$ is a part of the subverted/trigger set $\overline{\mathcal{X}}$.



Assume that $\mathcal{T}_{x,\mathsf{aux}}(x, \mathsf{aux})$ represents the communication transcript between the pool operator and the puzzle solving/mining device, given a puzzle input x and auxiliary information aux.

- $\mathsf{Exf}(sk, y, \mathcal{T}_{x,\mathsf{aux}}) \rightarrow \overline{\mathcal{T}}_{x,\mathsf{aux}}$: The exfiltration algorithm takes as input a secret key sk for symmetric encryption, a puzzle solution $y \in \mathcal{Y}$ and the communication transcript for the corresponding puzzle input, auxiliary information pair (x, aux). It outputs a $\overline{\mathcal{T}}_{x,\mathsf{aux}}$ to replace $\mathcal{T}_{x,\mathsf{aux}}$.
- $\mathsf{Ret}(sk, \overline{\mathcal{T}}_{x,\mathsf{aux}}) \rightarrow y/\perp$: The retrieve algorithm takes as input a secret key sk for symmetric encryption and the communication transcript $\overline{\mathcal{T}}_{x,\mathsf{aux}}$ for the corresponding puzzle input, auxiliary information pair (x, aux). It outputs a puzzle solution $y \in \mathcal{Y}$ or $\perp$.
- $\overline{\mathsf{P}}.\overline{\mathsf{Eval}}(sk, x, \mathsf{aux}, \overline{\mathcal{X}})$: The subverted puzzle evaluation algorithm takes a secret key sk for symmetric encryption, an input from the domain x, auxiliary information aux and a set $\overline{\mathcal{X}} \subset \mathcal{X}$.
  - If $x \in \mathcal{X} \setminus \overline{\mathcal{X}}$:
    - Compute a puzzle solution y_1.
    - Run the exfiltration algorithm $\mathsf{Exf}(sk, y_1, \mathcal{T}_{x,\mathsf{aux}})$.
    - Compute and output a puzzle solution y_2 and auxiliary information aux. Such that $y_2 \neq y_1$.
  - Else output a puzzle solution y and a proof π.

Fig. 6: The exfiltration procedure for our proposed leeching attack as referenced in Section C.2

D Background

Hashrate/Hashpower. In a proof-of-work based consensus network, for example, the network of Bitcoin miners, the *hashrate* of the network is the total number of SHA-256 hashes that can be calculated by all the devices participating in the network. This is an important metric because the difficulty of the Bitcoin proof-of-work is determined as a function of the network's hashrate [5]. It is adjusted every 2 weeks according to changes in the network hashrate. The hashrate of an individual miner then is the total number of hashes the individual miner (device) can compute. This again is an important metric because a proof-of-work based consensus is subject to a number of attacks when a single attacker controls a large fraction of the network's hashrate.

Attacker hashrate (δ)	RS Encoding Parameters RS(n, K)	Stateful Success Probability $1 - \frac{1}{(2-\delta)^n}\left(\sum_{i=1}^{K-1}\binom{n}{i}(1-\delta)^{n-i}\right)$	Stateless Success Probability
0.05	RS(520, 256)	0.83	$\mathsf{negl}(\lambda)$
0.10	RS(526, 256)	0.96	$\mathsf{negl}(\lambda)$
0.15	RS(533, 256)	0.997	$\mathsf{negl}(\lambda)$
0.20	RS(541, 256)	0.9997	$\mathsf{negl}(\lambda)$
0.25	RS(549, 256)	≈ 1	$\mathsf{negl}(\lambda)$

Table 2: Attack success probabilities and Reed-Solomon Encoding Parameters (with 1-bit symbols) for the stateful Load-shedding attack (in the blockchain input model), depending on attacker hashrate for a 256-bits trigger as referenced in Section C.3. Note that the attacker can have multiple triggers, but the number of triggers $\overline{\mathcal{X}}$ has to be negligible in terms of the security parameter to avoid detection. This also assumes that attacker hashrate never goes below δ for n consecutive blocks, changes will decrease probability of success.

51% attacks and Selfish Mining. A number of attacks have been proposed against proof-of-work blockchains. The most well-known is the 51% attack [38], which refers to a situation where a single party/cooperation has control of a majority of the network hashrate and can therefore "fork" and roll back transactions. A related attack called selfish mining [30] has the attacker mine a parallel competing blockchain fork, without publishing it. This attacker then continues to mine on this secret fork while the honest miners continue to mine on the previous "stale" version of the blockchain. The attacker can now judiciously make blocks from its secret branch public. This renders the computational effort of the honest miners moot, while also earning mining rewards for the attacker disproportionate to its hashrate. An attacker requires 1/3 of the network's hashrate to successfully mount such an attack.

Bitcoin block hashing algorithm. In the real world, the Bitcoin proof-of-work solution consists of the following fields [4]:

- Version (32-bits): Block version number
- hashPrevBlock (256-bits): Hash of the previous block
- hashMerkleRoot (256-bits): Merkle tree root of all the transactions to include in the proposed block
- time (32-bits): Proposed block timestamp as seconds since $1970-01-01\mathsf{T}00:00$ UTC
- diff (32-bits): Difficulty target
- nonce (32-bits): the nonce for which

  $$H(\mathsf{nonce}, H(\mathsf{hashMerkleRoot}, \mathsf{hashPrevBlock}))$$

  has the correct number of preceding $0'$s as specified by diff

E Cryptographic Puzzle Definitions

Definition 8 (Correctness). *A cryptographic puzzle is correct if* $\forall \lambda, \Delta$, $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta)$, *and* $\forall x \in \mathcal{X}$ *if* $y \leftarrow \mathsf{Eval}(x, \mathsf{aux})$ *then* $\mathsf{Verify}(x, \mathsf{aux}, y) = 1$.

Definition 9 (Soundness). *We require that an adversary can not get a verifier to accept an incorrect puzzle solution.*

$$\Pr\left[\begin{matrix} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta) \\ y \neq \mathsf{Eval}(x, \mathsf{aux}) \end{matrix} \middle| \begin{matrix} (x, y, \mathsf{aux}) \leftarrow \mathcal{A}(1^\lambda, \Delta, \mathsf{pp}) \\ \mathsf{Verify}(x, \mathsf{aux}, y) = 1 \end{matrix}\right]$$

$$\leq \mathsf{negl}(\lambda)$$

The difficulty parameter Δ defines resource requirements for finding a correct puzzle solution. Our resource-requirement definition 10 captures the said notion in real-world puzzle applications. Bitcoin's proof of work has a dynamic average-case resource requirement for puzzle evaluation. This is because the overall hashrate of the network keeps changing but the protocol changes difficulty/resource requirements such that the proof-of-work puzzle is solved once every 10 minutes. The best case resource requirement is trivial, there could be a bitcoin proof of work instance for which just computing one hash gives a valid solution. For VDFs and PoSpace on the other hand, the aim is to have the average-case and best-case requirement be the same. This gives a more uniform domain of puzzle instances with respect to resource requirement for evaluating solutions.

Definition 10 (Resource-requirement). *We require that no PPT adversary can solve the puzzle and consume less than* $\Theta_{avg}(\Delta)$ *of the puzzle resource while doing so in the average-case and* $\Theta_{best}(\Delta)$ *in the best-case.*

- ***Average-case.*** *Let* $\mathcal{R}^{avg}_{\mathcal{A}}(\Delta)$ *represent the average-case resource requirement for adversary* $\mathcal{A}$ *to solve a puzzle with difficulty parameter* Δ. *For a large number* n *of puzzle instances,* $\{x_1, x_2, \ldots, x_n\}$, *let indicator* $\mathbb{I} = 1$ *if* $\mathsf{Verify}(x_i, \mathsf{aux}_i, y_i) = 1 \forall i \in \{n\}$ *and* $\mathbb{I} = 0$ *otherwise.*

$$\Pr\left[\begin{matrix} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta), x_i \xleftarrow{\$} \mathcal{X} \\ (y_i, \mathsf{aux}_i) \leftarrow \mathcal{A}(1^\lambda, \Delta, \mathsf{pp}, x_i) \\ \mathcal{R}^{avg}_{\mathcal{A}}(\Delta) < \Theta_{avg}(\Delta) \\ \mathbb{I} = 1) \end{matrix}\right] \leq \mathsf{negl}(\lambda)$$

- ***Best-case.*** *Let* $\mathcal{R}^{best}_{\mathcal{A}}(\Delta)$ *represent the best-case resource requirement for adversary* $\mathcal{A}$ *to solve a puzzle with difficulty parameter* Δ. *It holds true* $\forall x \in \mathcal{X}$:

$$\Pr\left[\begin{matrix} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta), x \xleftarrow{\$} \mathcal{X} \\ (y, \mathsf{aux}) \leftarrow \mathcal{A}(1^\lambda, \Delta, \mathsf{pp}, x) \\ \mathsf{Verify}(x, \mathsf{aux}, y) = 1 \\ \mathcal{R}^{best}_{\mathcal{A}}(\Delta) < \Theta_{best}(\Delta) \end{matrix}\right] \leq \mathsf{negl}(\lambda)$$

Note that the pre-processing phase can be optional, in which case unprocessed and pre-processed input are the same and consequently $\mathcal{X}_{\mathsf{Pre}} = \mathcal{X}$ and the auxiliary information $\mathsf{aux} = \mathsf{nil}$. A similar worst-case resource requirement can be defined similarly.

F Proofs

Proof sketch for Theorem 2 If the puzzle pre-processing algorithm satisfies the unpredictability property then the adversary can only distinguish the input fed to the device from a random string with negligible probability. Therefore, the input to the device is effectively random for the adversary and hence it can feed the device a subverted input to the device with probability $\left|\overline{\mathcal{X}}/\mathcal{X}\right| = \mathsf{negl}(\lambda)$.

Proof for Theorem 3 We split the proof into two parts:

1. if the size of the set of subverted inputs $\left|\overline{\mathcal{X}}\right|$ is a non-negligible function of λ then the detector $\mathcal{D}$ detects the subverted inputs with non-negligible probability just by polynomial testing
2. if $\left|\overline{\mathcal{X}}\right| = \mathsf{negl}(\lambda)$ then the attacker $\mathcal{A}$ can not win the subversion game $\mathbf{Sub}^{\mathcal{A}}_{\mathsf{P},\bar{\mathsf{P}}}$ with non-negligible probability by Theorem 2

In either case, by definition 1, all cryptographic puzzle schemes $\mathsf{P} = (\mathsf{Setup}, \mathsf{Pre}, \mathsf{Eval}, \mathsf{Verify})$ utilizing a puzzle pre-processing algorithm satisfying the unpredictability property are secure against subversion via all input-induced attacks. □

Proof for Theorem 4 If the number of bits modified by the miner, before inputting the resulting hash to the mining device is some linear function $f(\lambda)$ in the security parameter, then probability of the pool operator correctly estimating the exact input to the mining device is $\mathsf{negl}(\lambda) = \frac{1}{2^{f(\lambda)}}$. Considering that the proof-of-work pre-processing as specified above allows upto 100 bytes of entropy to be used by the miners, it satisfies the unpredictability property as defined in definition 2.

G Further Analysis

G.1 Hardware Errors in Mining Devices

Hardware errors in mining devices reflect the number of inputs on which the mining ASIC fails to compute. Figure 1 shows how these errors appear on the mining software [41], with very little information on why the error occurred and on what particular input. We found this issue to be rampant across ASIC-based mining devices across networks such as Bitcoin, Litecoin etc. Following is a list of discussions from various forums on mining operations which report this as a concern, as evidence:

- From 2014-present, multiple reports (Bitcoin, Litecoin) on Reddit's mining forums on mining devices with hardware error rates ranging from a few every hour to a few every minute
- From a 2015 discussion on another bitcoin mining forum: Multiple users report high hardware error rate stats
- From April, 2020: High failure rates reported for latest Bitcoin mining devices on Cointelegraph [60]

G.2 Leeched Selfish-Mining Attack in Mining Pools

In proof-of-work mining, many miners pool their efforts with different mining pools. Each mining pool has its own software, and the pools have a *pool operator* who uses an algorithm to split work into *shares* that are farmed out to workers. Let the block reward minus the fee charged by the pool operator be B. Each worker then is paid reward $R = B \cdot \frac{n}{N}$, where n is fraction of work their shares represent and N is the total amount of work represented by all shares. To prove that workers are contributing, they must output solutions found at much lower difficulty levels (this assumes a puzzle that can find low-difficulty solutions while also searching for high-difficulty solutions, a property that is not common to all puzzles.) These low-difficulty solutions are used as a subliminal channel for exfiltration. Figure 6 presents the exact exfiltration process. As an example, the following attack strategy can be employed by a Bitcoin mining pool operator who also manufactures mining hardware or colludes with the manufacturer. Let the set of devices manufactured by this manufacturer be D, a subset $\overline{\mathsf{D}}$ of D are subverted.

- Embed the subverted set of inputs $\overline{\mathcal{X}}$ in the hardware during manufacturing.
- Use the work shares (low-difficulty solutions) as a secure exfiltration channel E
- On receiving an exfiltrated puzzle solution, follow the selfish-mining attack strategy and secretly mine on this fork of the blockchain on devices in the set $\mathsf{D} \setminus \overline{\mathsf{D}}$.

The following is a concrete method to utilize the malicious pool operator and the subverted hardware for exfiltration:

- Given m work shares, utilize them to communicate a n-bit string
- Submit work shares such that the low-order bit of the i'th work share is also the i'th bit of the (encoded, see Figure 6) puzzle solution

All the bits of the solution can be leaked even if ≈ 300 work shares are used. As a point of reference, consider that all Bitcoin mining pools currently set the work share difficulty for each device such that one work share is generated every 2-3 seconds [49]. Given the Bitcoin block time of 10 minutes, each miner is sending up to 300 work shares per block to its pool operator. *This amount of communication is enough to exfiltrate all the bits of a proof of work, encrypted or unencrypted.*

G.3 Malleability of Blockchain's Entropy

We adopt the model of an adversary who malleates a blockchain's entropy from the work of Pierrot and Wesolowski [42]. This model is specifically applicable to a proof of work based blockchain. They first start by assuming that the underlying hash function used for proof of work is secure. For d being the difficulty of the proof of work, each new block contains d bits of computational min-entropy, this was first justified in [20]. We know that there exists a cryptographic extractor [27] Ext, which maps each block's d bits of min-entropy to $\lfloor d/2 \rfloor$ of near-uniform bits.

The adversary's attempt to mine favorable blocks is then captured by the following game played between a set of honest miners and an adversary $\mathcal{A}$. The goal of the adversary is to bias the probability distribution of the extract of the *decisive block*, which is a future block of interest to the adversary. The adversary holds influence over a δ fraction of the blockchain. For $s_{\mathcal{A}}$ being the number of hashes per second the adversary can calculate and s being the number of hashes per second all miners combined (including $\mathcal{A}$) can calculate,

$$\delta = \frac{s_{\mathcal{A}}}{s}$$

The game starts when a fixed *initial block*, indexed 0 is received and ends when block height $n + f$ is reached. Here n is the block height of the decisive block and it takes f additional blocks to finalize the decisive block. This is to ensure that there are no forks of the blockchain which would alter the decisive block. The adversary wins if the extract x of the decisive block B, $\mathsf{Ext}(B) = x$ falls in a subset favorable to the adversary. We denote this via the indicator function, $\mathbb{I}$ where $\mathbb{I}(x) = 1$ if $x \in \mathcal{F}$ and $\mathbb{I}(x) = 0$ otherwise, for $\mathcal{F}$ being the favorable set for the adversary.

Our interest lies in the setting where the extractor Ext is private and set by the adversary $\mathcal{A}$. The extractor values are used by the adversary who mounts the load shedding attack as a trigger. The load shedding adversary therefore uses the blockchain to signal the mining evaluation hardware/software to malfunction.

G.4 On Load-shedding

Challenges in the blockchain input model. In this setting, if an attacker aims to mount a load-shedding attack, they leverage their influence over the blockchain. Therefore, we first focus on the probability with which such an adversary can bias the blockchain's inputs based on their bias ability. Assuming the puzzle solvers start solving for a solution as soon as the next potential block is broadcast, the adversary's goal is to broadcast a block favorable to it. In the process, the adversary might throw away some a valid puzzle solutions it finds. The variable μ represents the favorable set $\mathcal{F}$ as a fraction of the puzzle domain size. From the work of Pierrot and Wesolowski [42, Section 3.3], we borrow the following result which gives us the probability with which as adversary $\mathcal{A}$ with δ-bias ability wins the $\delta^{\mathcal{A}}_{\mathsf{bias}}$ game:

$$\Pr\left[\delta^{\mathcal{A}}_{\mathsf{bias}} = 1\right] = \sum_{a>0}(\delta(1-\mu))^a\mu = \frac{\mu}{1-\delta(1-\mu)}$$

Similar to the cited work, we note that when $\delta \geq 0$ and $\mu \leq 1$, the above probability is better than μ. As is clear from the calculation above, when μ is negligibly small compared to the security parameter, this attack does not work very well.

Stateful Load-shedding. The goal of the adversary is to bias the blockchain such that the extract of the decisive block $\mathsf{Ext}(B) = 1$, here $\mathsf{Ext}(B) = 1$ if the low-order bit of the block header is 1. Assuming an error-correcting code such that given a message length of k-bits and some failure rate p, it encodes the message in n-bits such that even if $p.k$ bits of the encoding get flipped, the message can still be decoded successfully. Therefore, for the attacker to successfully trigger a stateful load-shedding attack, it needs to win the following $\delta^{\mathcal{A}}_{\mathsf{bias}}$ game (K)-out-of-n times (where $K = n - p.k$). As a result of this strategy, probability of subversion increases drastically while probability of detection does not change at all. This happens with probability 1/2, therefore the favorable set of the adversary is 1/2 of the possible outcomes. This gives us that $\mu = 1/2$. Let $P_{\mathcal{A}}$ be the win probability in the $\delta^{\mathcal{A}}_{\mathsf{bias}}$ game as defined above,

$$P_{\mathcal{A}} = \frac{\mu}{1-\delta(1-\mu)} = \frac{\frac{1}{2}}{1-\frac{\delta}{2}} = \frac{1}{2-\delta} \text{ as } \mu = \frac{1}{2}$$

Therefore, given that the stateful load-shedding attack succeeds if the adversary wins the above game at least (K)-out-of-n times (where $K = n - p.k$). Let the string $s \in \{0,1\}$ represent the results of n consecutive $\delta^{\mathcal{A}}_{\mathsf{bias}}$ games played by the adversary. The i-th bit of s represents the result of the i-th $\delta^{\mathcal{A}}_{\mathsf{bias}}$ game. Let s_1 represent the number of 1s in s, and s_0 represent the number of 0s in s. The attack success probability:

$$\Pr[s_1 \geq K] = \Pr[s_0 < K]$$

$$= 1 - \left(\sum_{i=1}^{K-1}\binom{n}{i}\left(\frac{1}{2-\delta}\right)^i\left(\frac{1-\delta}{2-\delta}\right)^{n-i}\right)$$

$$= 1 - \frac{1}{(2-\delta)^n}\left(\sum_{i=1}^{K-1}\binom{n}{i}(1-\delta)^{n-i}\right)$$

G.5 Test-friendly proof-of-work

A simple example of a test-friendly puzzle for Bitcoin includes the description of a modified programmable hash function $H_{r,p}$ as part of the puzzle instance. This function has the simple property that $H(r) = p$, where p is a valid puzzle solution and r is some input that a puzzle solver will identify in ρ units of time using a known test strategy. Such puzzles are relatively easy to construct using standard

hash functions: simply define $H_{r,p}(x) = H(x) \oplus T$ where T is an additional string with length equal to the hash function's range. To program a specific pair r, p the tester simply computes $T \leftarrow H(r) \oplus p$. It is easy to see that if H is modeled as a random oracle and T is chosen uniformly at random, the resulting function produces output that is statistically identical to a random oracle. Hence, provided that a fresh T is sampled uniformly by a network every few days, such instances can be used safely as valid puzzle instances. Consequently, the implementations should provide T as an input parameter. We omit a detailed analysis of this approach, but we note that it is similar to a proposal by Russell *et al.* [48] and benefits from the same analysis.

G.6 Ethereum's Beacon Chain

Ethereum plans on using VDFs as a source of unpredictable randomness. An example of this requirement in Ethereum 2.0 is the following. A small of group of validators are required to progressively build a chain of randomness, this chain is termed the "Beacon-chain"[28]. Assuming a global clock and splitting time into contiguous 8-second blocks and 128-slot epochs, one value O is generated per epoch $\mathcal{E}$. This generated random value is used to select a validator who then gets the opportunity to propose the next block to be added to the blockchain and consequently reaps the block reward. In the ideal scenario, with unbiased randomness the frequency with which a validator is selected is directly proportional their stake in the system. However, if a malicious actor is successful in biasing the randomness they can sample strings such that they can select the one which benefits them most. Currently the randomness O is obtained from the reveals of a RANDAO commit-reveal scheme used to generate a random number where the commits are inputs produced by the validators during epoch $\mathcal{E}$. In a RANDAO commit-reveal scheme, every beacon chain proposer is committed to 32 bytes of local entropy. (In practice a chain of commit-reveals is setup with a hash onion for validator registration.) Beacon chain proposers may reveal their local entropy by extending the canonical beacon chain with a block. Honest proposers are expected to keep their local entropy private until their assigned slot. The beacon chain maintains 32 bytes of on-chain entropy by XOR-ing the local entropy revealed at every block. However such a commit-reveal scheme is biasable: a malicious validator that controls the last reveal can choose to reveal or not, giving them some control over the choice of O based on their decision. Therefore, one proposed way to make O unbiasable is to pass O through a *verifiable delay function* (VDF) which is guaranteed to be slow to compute. This is done so that all validators must choose whether to reveal or not *before* they know the output of the VDF.

Let $B(\cdot, b_i)$ be a function which produces biasable randomness on input an epoch i and the 32 bytes of local entropy b_i from the beacon chain proposer selected for this round. Then the randomness beacon $R(\cdot)$ output for epoch i is computed as follows:

$$B(i, b_i) = B(i - 1, b_{i-1}) \oplus b_i$$

$$R(i) = \mathsf{VDF.Eval}(pp, B(i-T, b_{i-T}))$$

Assuming it takes real world time T for the commodity hardware to compute the VDF output. This construction ensures that the VDF input is available at the same time to all parties interesting in computing its output. The Ethereum foundation may spend over $15 million [36] for the development of specialized hardware to provide to the validators, in order to achieve lowest evaluation time for any VDF parameters.

G.7 Masked Halving Correctness.

The only change to the original halving protocol by the masked halving protocol is that instead of directly computing $y = x^{2^T}$, masked halving computes $x_b = x.b$ and then computes $y_b = x_b^{2^T}$. To unmask the final result, we compute $y = y_b.(b^{2^T})^{-1}$. If b was a previous instance then b^{2^T} is known already. Furthermore, during the evaluation of b^{2^T} the intermediate steps can be stored in memory as $\alpha = \{b^{2^T}, b^{2^{T/2}}, \ldots, b^2\}$. The terms in $\beta = \{(b^{2^T})^{-1}, (b^{2^{T/2}})^{-1}, \ldots, (b^2)^{-1}\}$ can be pre-computed by the evaluator. In each halving now, to compute μ the evaluator computes $x_b^{2^{T/2}}.(b^{2^{T/2}})^{-1} = x^{2^{T/2}}$ which is the value of μ in the original halving protocol. This is the only change in the halving process, making masked halving correct and minimally taxing.

G.8 Real-world Cryptographic Puzzles

Definition 11 (Bitcoin Proof of Work as a Cryptographic Puzzle). *Bitcoin proof of work can be formalized as a cryptographic puzzle as follows:*

- $\mathsf{Setup}(1^\lambda, \Delta) \to \mathsf{pp}$*: The security parameter* 1^λ *contains the specifications for the hash function (SHA-256) used for Bitcoin' proof of work including the number of bits of security it provides. The difficulty parameter is an integer which indicates minimum amount of leading zeroes required for a valid proof of work solution.*
- $\mathsf{Pre}(x', \mathsf{aux}) \to x$*: The unprocessed puzzle input* x' *is the previous block header/proof of work. The auxiliary input* aux *is the Merkle tree root of the transactions confirmed and included in the proposed block. The pre-processed input is computed as* $x = H(\mathsf{aux}, x')$*.*
- $\mathsf{Eval}(x, \mathsf{aux}) \to y/\perp$*: The evaluation algorithm checks if for any input* r *in the nonce range, the hash* $H(r, x)$ *has* $\geq \Delta$ *many preceding zeroes. If it finds such a solution it outputs* $y = r$*, otherwise outputs* $\perp$*.*
- $\mathsf{Verify}(x, \mathsf{aux}, y) \to 0/1$*: If* $H(y, x)$ *has* $\geq \Delta$ *many preceding zeroes, output* 1 *and* 0 *otherwise.*

It is worth noting that in the real world, the pre-processing phase can possibly be completely predictable. Typical bitcoin mining logic is to include transactions with the highest miner fees. If this is the case then the input to the Eval algorithm

become predictable. Since the inputs to Eval are the inputs to the mining rig, this opens up the possibility of input-induced attacks which we discuss in Sections C and C.3. However, this does not imply that Bitcoin proof of work is a bad cryptographic puzzle construction. This particular issue can be easily fixed by introducing some entropy into the process of selecting transactions.

Definition 12 (Verifiable Delay Functions). *A verifiable delay function (VDF) consists of the following algorithms:*

- $\mathsf{Setup}(1^\lambda, \Delta) \rightarrow \mathsf{pp}$*: The VDF generation algorithm takes as input a security parameter* 1^λ *and a difficulty parameter* Δ *and outputs public parameters* pp *which fix the domain* $\mathcal{X}$ *and range* $\mathcal{Y}$ *of the VDF and other information required to compute a VDF or verify a solution.*
- $\mathsf{Eval}(\mathsf{pp}, x) \rightarrow (y, \pi)$*: The VDF evaluation algorithm takes as input the public parameters* pp*, an input from the domain* x*. It outputs a VDF solution* y *and a proof* π*.*
- $\mathsf{Verify}(\mathsf{pp}, x, y, \pi) \rightarrow 0/1$*: The VDF verification algorithm takes as input the public parameters* pp*, an input from the domain* x*, an input from the range* y *and a proof* π*. It outputs either* 0 *or* 1*.*

Additionally, VDFs must satisfy the correctness, soundness and sequentiality definitions as defined below.

Definition 13 (Correctness). *A verifiable delay function is correct if* $\forall \lambda, \Delta$*,* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta)$*, and* $\forall x \in \mathcal{X}$ *if* $(y, \pi) \leftarrow \mathsf{Eval}(\mathsf{pp}, x)$ *then* $\mathsf{Verify}(\mathsf{pp}, x, y, \pi) = 1$*.*

Definition 14 (Soundness). *We require that an adversary can not get a verifier to accept an incorrect VDF solution.*

$$\Pr\left[\begin{matrix}\mathsf{Verify}(\mathsf{pp}, x, y, \pi) = 1 \\ y \neq \mathsf{Eval}(\mathsf{pp}, x)\end{matrix} \middle| \begin{matrix}\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \Delta) \\ (x, y, \pi) \leftarrow \mathcal{A}(1^\lambda, \Delta, \mathsf{pp})\end{matrix}\right] \leq \mathsf{negl}(\lambda)$$

Definition 15 (VDF as a Cryptographic Puzzle). *A VDF can be formalized as a cryptographic puzzle as follows:*

- $\mathsf{Setup}(1^\lambda, \Delta) \rightarrow \mathsf{pp}$*: This algorithm has inputs and outputs exactly similar to as described in* $\mathsf{VDF.Setup}()$*.*
- $\mathsf{Pre}(x', \mathsf{aux}) \rightarrow x$*: Set* $\mathsf{aux} = \mathsf{nil}$ *and* $x = x'$*.*
- $\mathsf{Eval}(x, \mathsf{aux}) \rightarrow y/\perp$*: The evaluation algorithm proceeds exactly as described in* $\mathsf{VDF.Eval}()$*. The output* y *is a tuple consisting of a VDF solution* s *and its proof* π*,* $y = (s, \pi)$*.*
- $\mathsf{Verify}(x, \mathsf{aux}, y) \rightarrow 0/1$*: Outputs the result from computing* $\mathsf{VDF.Verify}(\mathsf{pp}, x, s, \pi)$*.*

VDF security is described using a sequentiality game played between a challenger and an adversary as defined below:

Definition 16 (Sequentiality). *The sequentiality game captures the notion that no adversary should be able to compute the output for* Eval *on a random challenge in time less than the requisite time t even with arbitrary parallelism. For an exact description of the game and more details we refer readers to Boneh et al's [19] work.*

Definition 17 (Proof of Space). *A proof of space is defined using the following algorithms:*

- $\mathsf{Init}(N, pk) \to S$*: The initialization algorithm takes as input a space parameter $N \in \mathcal{N}$ (where $\mathcal{N} \subset \mathbb{Z}^+$ is the set of valid parameters) and a public key pk for a signature scheme. It outputs $S = (S.\Lambda, S.N, S.pk)$ where S consists of the space taxing file the prover needs to store.*
- $\mathsf{Prove}(S, c) \to \pi$*: The prove algorithm on an input S and a challenge $c \in \{0,1\}^w$ in the challenge space, outputs a proof π.*
- $\mathsf{Verify}(S, c, \pi) \to 0/1$*: The verify algorithm accepts the proof and outputs* 1 *if it is a valid proof of space for the given input, challenge pair. Otherwise, it outputs* 1*.*

Additionally, a PoSpace must satisfy the completeness and security definitions as defined below.

Definition 18 (PoSpace Completeness). *Perfect completeness implies that* $\forall N \in \mathcal{N}, c \in \{0,1\}^w$,

$$\Pr[\mathsf{Verify}(S, c, \pi) = 1] = 1$$

where $S \leftarrow \mathsf{Init}(N, pk)$ *and* $\pi \leftarrow \mathsf{Prove}(S, c)$.

Definition 19 (PoSpace Security). *Informally, proof of space security states that an adversary who stores a file of size considerably less than N bits should not be able to produce a valid proof when given a random challenge without using a significant amount of computation. For an exact description of the game and more details we refer readers to [29,8].*

H Practical Concerns

Beyond the attacks we propose and analyze, the following are real-world concerns relevant to puzzle-solving devices:
Limits of testing. Previous work [11] models the oracles in the detection game as instantaneous response oracles. While they point out that the goal of their model is not to evade all forms of detection, these factors are crucial for real-world attack considerations. Keeping that in mind, our attacks still work in the model where the detector considers oracle response times. However, there are out of model attacks which still can not be detected via our testing regime.
Very strong attackers. Such attackers can come up with numerous attack strategies which are not input-induced. For example:

- **Simple timer-based attack.** The attacker fits a small clock in the puzzle solving device. After a certain time T, the device's puzzle solving ability drops by some fraction. Notice that such drops are explained as device aging etc. It is not clear how this attack can be detected in the real world. Another attack in a similar vein is one where the device's puzzle solving ability drops after computing a certain number of puzzle solutions (at any difficulty level).
- **Subverting only real-world difficulty parameters.** If we use the example of proof of work mining. The testing of mining devices currently includes testing the number of hashes computed per second and checking solutions outputted at lower difficulty levels. An attacker can ensure that the device is subverted only on puzzle instances of difficulty levels relevant in real-world applications. The detector could then try to verify the devices behavior on puzzles at real-world difficulty parameters where the solution is already known, for example, known bitcoin blocks (by feeding in a small nonce range and Merkle root of the proposed block). The attacker can easily counter such detection by ensuring the device never fails on existing blocks since this is a set of a few hundred thousand puzzle instances. The detector is then only left with the option to test the device on the freshly mined blocks available after the purchase of the device.

Stronger Input-induced attacks. The attacks we describe and analyze throughout the paper are input-induced attacks where the trigger inputs are a negligible fraction of the puzzle input domain. This is mainly to ensure that a PPT detector can detect all attacks which are triggered by a higher fraction of the puzzle input domain. An attacker might adopt a strategy where it has a long-enough time window to reap the benefits of the subversion before being detected. At the point of detection, the attacker has already succeeded and made a huge profit.
Real-world detection costs. We model our detector as a PPT party that detects all input-induced attacks except ones that succeed on a negligibly small set of inputs. However, there are many other factors to consider in the real world. Polynomial sampling might result in detection but implies a costly testing period in the real world. The puzzle solver is losing valuable evaluation time every testing cycle. This economic disincentivization of testing means that an attacker can increase its input-induced attack surface in the real world. There are also device life cycles to consider, proof-of-work mining hardware has a life-cycle <2 years [3,37]. The attacker can mount attacks that are active only once in the life-cycle of the hardware. Our analysis (Fig 5) shows that unless a miner has over a hundred thousand dollars worth of mining capacity, they cannot statistically test even against the attacks we propose. This indicates that certain subversions, such as the ones that only affect the device a few times in its lifetime, are nearly impossible to detect. To that end, there is a huge need for such devices to be produced in facilities where the manufacturing process can be scrutinized. Creating processes to manufacture backdoor-resistant devices in that setting has already been studied in existing work [56,18]. However, today there are billions of dollars worth of puzzle-solving devices already sold and used without such guarantees. We hope our work serves as a warning to prevent leaving such a

large attack surface area in the future design of blockchain consensus protocols using cryptographic hardware.